6502 Assembly Language Subroutines

Lance A. Leventhal
Winthrop Saville

OSBORNE/McGraw-Hill
Berkeley, California

Disclaimer of Warranties and Limitation of Liabilities

The authors have taken due care in preparing this book and the programs in it, including research, development, and testing to ascertain their effectiveness. The authors and the publishers make no expressed or implied warranty of any kind with regard to these programs nor the supplementary documentation in this book. In no event shall the authors or the publishers be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance, or use of any of these programs.

Apple II is a trademark of Apple Computer, Inc.

Published by
Osborne/ McGraw-Hill
2600 Tenth St.
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write OSBORNE/ McGraw-Hill at the above address.

6502 ASSEMBLY LANGUAGE SUBROUTINES

34567890 DODO 876543

ISBN 0-931988-59-4

Cover art by Jean Frega.

Text design by Paul Butzler.

Contents

Preface

This book is intended to serve as a source and a reference for the assembly language programmer. It contains an overview of assembly language programming for a particular microprocessor and a collection of useful routines. In writing the routines, we have used a standard format, documentation package, and parameter passing techniques. We have followed the rules of the original manufacturer's assembler and have described the purpose, procedure, parameters, results, execution time, and memory usage of each routine.

This overview of assembly language programming provides a summary for those who do not have the time or need for a complete textbook such as is provided already in the Assembly Language Programming series. Chapter 1 contains an introduction to assembly language programming for the particular processor and a brief summary of the major features that differentiate this processor from other microprocessors and minicomputers. Chapter 2 describes how to implement instructions and addressing modes that are not explicitly available. Chapter 3 discusses common errors that the programmer is likely to encounter.

The collection of routines emphasizes common tasks that occur in many applications such as code conversion, array manipulation, arithmetic, bit manipulation, shifting functions, string manipulation, summation, sorting, and searching. We have also provided examples of I/O routines, interrupt service routines, and initialization routines for common family chips such as parallel interfaces, serial interfaces, and timers. You should be able to use these routines as subroutines in actual applications and as guidelines for more complex programs.

We have aimed this book at the person who wants to use assembly language immediately, rather than just learn about it. The reader could be

- An engineer, technician, or programmer who must write assembly language programs for use in a design project.
- A microcomputer user who wants to write an I/O driver, a diagnostic program, or a utility or systems program in assembly language.

- A programmer or engineer with experience in assembly language who needs a quick review of techniques for a particular microprocessor.
- A system designer or programmer who needs a specific routine or technique for immediate use.
- A programmer who works in high-level languages but who must debug or optimize programs at the assembly level or must link a program written in a high-level language to one written in assembly language.
- A system designer or maintenance programmer who must quickly understand how specific assembly language programs operate.
- A microcomputer owner who wants to understand how the operating system works on a particular computer, or who wants to gain complete access to the computer's facilities.
- A student, hobbyist, or teacher who wants to see some examples of working assembly language programs.

This book can also serve as supplementary material for students of the Assembly Language Programming series.

This book should save the reader time and effort. There is no need to write, debug, test, or optimize standard routines, nor should the reader have to search through material with which he or she is thoroughly familiar. The reader should be able to obtain the specific information, routine, or technique that he or she needs with a minimum amount of effort. We have organized and indexed this book for rapid use and reference.

Obviously, a book with such an aim demands response from its readers. We have, of course, tested all the programs thoroughly and documented them carefully. If you find any errors, please inform the publisher. If you have suggestions for additional topics, routines, programming hints, index entries, and so forth, please tell us about them. We have drawn on our programming experience to develop this book, but we need your help to improve it. We would greatly appreciate your comments, criticisms, and suggestions.

NOMENCLATURE

We have used the following nomenclature in this book to describe the architecture of the 6502 processor, to specify operands, and to represent general values of numbers and addresses.

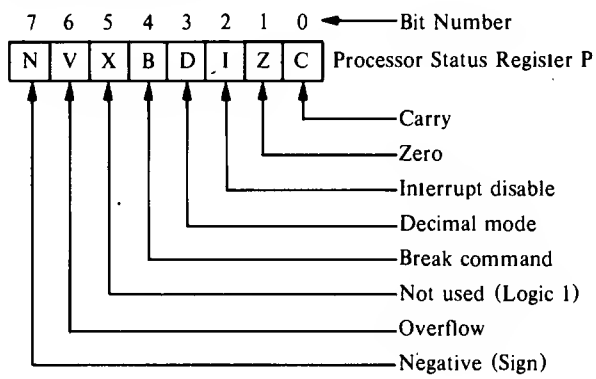
6502 Architecture

Byte-length registers include

A (accumulator)

F (flags, same as P)
P (status register)
S or SP (stack pointer)
X (index register X)
Y (index register Y)

Of these, the general purpose user registers are A, X, and Y. The stack pointer always contains the address of the next available stack location on page 1 of memory (addresses 0100_{16} through $01FF_{16}$). The P (status) or F (flag) register consists of a set of bits with independent functions and meanings, organized as shown in the following diagram:



Word-length registers include

PC (program counter)

Note: Pairs of memory locations on page 0 may also be used as word-length registers to hold indirect addresses. The lower address holds the less significant byte and the higher address holds the more significant byte. Since the 6502 provides automatic wraparound, addresses $00FF_{16}$ and 0000_{16} form a rarely used pair.

Flags include

Break (B)
Carry (C)
Decimal Mode (D)
Interrupt Disable (I)
Negative or Sign (N)
Overflow (V)
Zero (Z)

These flags are arranged in the P or F register as shown previously.

6502 Assembler

Delimiters include

space	After a label or an operation code
,	Between operands in the operand (address) field
;	Before a comment
:	After a label (optional)
(,)	Around an indirect address

Pseudo-Operations include

.BLOCK	Reserve bytes of memory; reserve the specified number of bytes of memory for temporary storage
.BYTE	Form byte-length data; place the specified 8-bit data in the next available memory locations
.DBYTE	Form double-byte (word) length data with more significant byte first; place the specified 16-bit data in the next available memory locations with more significant byte first
.END	End of program
.EQU	Equate; define the attached label
.TEXT	Form string of ASCII characters; place the specified ASCII characters in the next available memory locations
.WORD	Form double-byte (word) length data with less significant byte first; place the specified 16-bit data in the next available memory locations with less significant byte first
*=	Set origin; assign the object code generated from the subsequent assembly language statements to memory addresses starting with the one specified
=	Equate; define the attached label

Designations include

Number systems:

$ (prefix) or H (suffix)	Hexadecimal
@ (prefix) or Q (suffix)	Octal
% (prefix) or B (suffix)	Binary

The default mode is decimal.

Others:

' (in front of character)	ASCII
*	Current value of location (program) counter

' ' or " " (around a string of characters)	ASCII string
#	Immediate addressing
,x	Indexed addressing with index register X
,y	Indexed addressing with index register Y

The default addressing mode is absolute (direct) addressing.

General Nomenclature

ADDR	a 16-bit address in data memory
ADDRH	the more significant byte of ADDR
ADDRL	the less significant byte of ADDR
BASE	a constant 16-bit address
BASEH	the more significant byte of BASE
BASEL	the less signficant byte of BASE
DEST	a 16-bit address in program memory, the destination for a jump or branch instruction
NTIMES	an 8-bit data item
NTIMH	an 8-bit data item
NTIMHC	an 8-bit data item
NTIML	an 8-bit data item
NTIMLC	an 8-bit data item
OPER	a 16-bit address in data memory
OPER1	a 16-bit address in data memory
OPER2	a 16-bit address in data memory
PGZRO	an address on page 0 of data memory
PGZRO+1	the address one larger than PGZRO (with no carry to the more significant byte)
POINTER	a 16-bit address in data memory
POINTH	the more significant byte of POINTER
POINTL	the less significant byte of POINTER
RESLT	a 16-bit address in data memory

VAL16	a 16-bit data item
VAL16L	the less significant byte of VAL16
VAL16M	the more significant byte of VAL16
VALUE	an 8-bit data item
ZCOUNT	a 16-bit address in data memory

Chapter 1 General Programming Methods

This chapter describes general methods for writing assembly language programs for the 6502 and related microprocessors. It presents techniques for performing the following operations:

- Loading and saving registers
- Storing data in memory
- Arithmetic and logical functions
- Bit manipulation
- Bit testing
- Testing for specific values
- Numerical comparisons
- Looping (repeating sequences of operations)
- Array processing and manipulation
- Table lookup
- Character code manipulation
- Code conversion
- Multiple-precision arithmetic
- Multiplication and division
- List processing
- Processing of data structures.

Special sections discuss passing parameters to subroutines, writing I/O drivers and interrupt service routines, and making programs run faster or use less memory.

The operations described are required in applications such as instrumentation, test equipment, computer peripherals, communications equipment, industrial control, process control, aerospace and military systems, business equipment,

and consumer products. Microcomputer users will make use of these operations in writing I/O drivers, utility programs, diagnostics, and systems software, and in understanding, debugging, or improving programs written in high-level languages. This chapter provides a brief guide to 6502 assembly language programming for those who have an immediate application in mind.

QUICK SUMMARY FOR EXPERIENCED PROGRAMMERS

For those who are familiar with assembly language programming on other processors, we provide here a brief review of the peculiarities of the 6502. Being aware of these unusual features can save you a great deal of time and trouble.

1. The Carry flag acts as an inverted borrow in subtraction. A Subtract (SBC) or Compare (CMP, CPX, or CPY) instruction clears the Carry if the operation requires a borrow and sets it if it does not. The SBC instruction accounts for this inversion by subtracting 1−Carry from the usual difference. Thus, the Carry has the opposite meaning after subtraction (or comparison) on the 6502 than it has on most other computers.

2. The only Addition and Subtraction instructions are ADC (Add with Carry) and SBC (Subtract with Carry). If you wish to exclude the Carry flag, you must clear it before addition or set it before subtraction. That is, you can simulate a normal Add instruction with

```
CLC
ADC     MEMORY
```

and a normal Subtract instruction with

```
SEC
SBC     MEMORY
```

3. There are no 16-bit registers and no operations that act on 16-bit quantities. The lack of 16-bit registers is commonly overcome by using pointers stored on page 0 and the indirect indexed (postindexed) addressing mode. However, both initializing and changing those pointers require sequences of 8-bit operations.

4. There is no true indirect addressing except with JMP. For many other instructions, however, you can simulate indirect addressing by clearing index register Y and using indirect indexed addressing, or by clearing index register X and using indexed indirect addressing. Both of these modes are limited to indirect addresses stored on page 0.

5. The stack is always on page 1 of memory. The stack pointer contains the less significant byte of the next empty address. Thus, the stack is limited to 256 bytes of memory.

6. The JSR (Jump to Subroutine) instruction saves the address of its own third byte in the stack, that is, JSR saves the return address minus 1. RTS (Return from Subroutine) loads the program counter from the top of the stack and then adds 1 to it. You must remember this offset of 1 in debugging and using JSR or RTS for purposes other than ordinary calls and returns.

7. The Decimal Mode (D) flag is used to perform decimal arithmetic. When this flag is set, all additions and subtractions produce decimal results. Increments and decrements, however, produce binary results regardless of the mode. The problem with this approach is that you may not be sure of the initial or current state of the D flag (the processor does not initialize it on Reset). A simple way to avoid problems in programs that use Addition or Subtraction instructions is to save the original D flag in the stack, assign D the appropriate value, and restore the original value before exiting. Interrupt service routines, in particular, should always either set or clear D before executing any addition or subtraction instructions. The PHP (Store Status Register in Stack) and PLP (Load Status Register from Stack) instructions can be used to save and restore the D flag, if necessary. The overall system startup routine must initialize D (usually to 0, indicating binary mode, with CLD). Most 6502-based operating systems assume the binary mode as a default and always return to that mode as soon as possible.

A minor quirk of the 6502's decimal mode is that the Zero and Negative flags are no longer universally valid. These flags reflect only the binary result, not the decimal result; only the Carry flag always reflects the decimal result. Thus, for example, subtracting 80_{16} from 50_{16} in the decimal mode sets the Negative flag (since the binary result is $D0_{16}$), even though the decimal result (70_{16}) has a most significant bit of 0. Similarly, adding 50_{16} and 50_{16} in the decimal mode clears the Zero flag (since the binary result is $A0_{16}$), even though the decimal result is zero. Note that adding 50_{16} and 50_{16} in the decimal mode does set the Carry. Thus when working in the decimal mode, the programmer should use only branches that depend on the Carry flag or operations that do not depend on the mode at all (such as subtractions or comparisons followed by branches on the Zero flag).

8. Ordinary Load (or Pull from the Stack) and Transfer instructions (except TXS) affect the Negative (Sign) and Zero flags. This is not the case with the 8080, 8085, or Z-80 microprocessors. Storing data in memory does not affect any flags.

9. INC and DEC cannot be applied to the accumulator. To increment A, use

```
CLC
ADC     #1              ;INCREMENT ACCUMULATOR BY 1
```

To decrement A, use

```
SEC
SBC     #1              ;DECREMENT ACCUMULATOR BY 1
```

10. The index registers are only 8 bits long. This creates obvious problems in handling arrays or areas of memory that are longer than 256 bytes. To overcome this, use the indirect indexed (postindexed) addressing mode. This mode allows you to store the starting address of the array in two memory locations on page 0. Whenever the program completes a 256-byte section, it must add 1 to the more significant byte of the indirect address before proceeding to the next section. The processor knows that it has completed a section when index register Y returns to 0. A typical sequence is

```
INY                 ;PROCEED TO NEXT BYTE
BNE     LOOP        ;UNLESS A PAGE IS DONE
INC     INDR+1      ;IF ONE IS, GO ON TO THE NEXT PAGE
```

Memory location INDR+1 (on page 0) contains the most significant byte of the indirect address.

11. 16-bit counters may be maintained in two memory locations. Counting up is much easier than counting down since you can use the sequence

```
INC     COUNTL      ;COUNT UP LESS SIGNIFICANT BYTE
BNE     LOOP
INC     COUNTH      ;CARRYING TO MSB IF NECESSARY
JMP     LOOP
```

COUNTL contains the less significant byte of a 16-bit counter and COUNTH the more significant byte. Note that we check the Zero flag rather than the Carry flag since, as on most computers, Increment and Decrement instructions do not affect Carry.

12. The BIT instruction (logical AND with no result saved) has several unusual features. In the first place, it allows only direct addressing (absolute and zero page). If you want to test bit 3 of memory location ADDR, you must use the sequence

```
LDA     #%00001000
BIT     ADDR
```

BIT also loads the Negative and Overflow flags with the contents of bits 7 and 6 of the memory location, respectively, regardless of the value in the accumulator. Thus, you can perform the following operations without loading the accumulator at all. Branch to DEST if bit 7 of ADDR is 1

```
BIT     ADDR
BMI     DEST
```

Branch to DEST if bit 6 of ADDR is 0

```
BIT     ADDR
BVC     DEST
```

Of course, you should document the special use of the Overflow flag for later reference.

13. The processor lacks some common instructions that are available on the 6800, 6809, and similar processors. Most of the missing instructions are easy to simulate, although the documentation can become awkward. In particular, we should mention Clear (use load immediate with 0 instead), Complement (use logical EXCLUSIVE OR with the all 1s byte instead), and the previously mentioned Add (without carry) and Subtract (without borrow). There is also no direct way to load or store the stack pointer (this can be done through index register X), load or store the status register (this can be done through the stack), or perform operations between registers (one must be stored in memory). Other missing instructions include Unconditional Relative Branch (use jump or assign a value to a flag and branch on it having that value), Increment and Decrement Accumulator (use the Addition and Subtraction instructions), Arithmetic Shift (copy bit 7 into Carry and rotate), and Test zero or minus (use a comparison with 0 or an increment, decrement sequence). Weller[1] describes the definition of macros to replace the missing instructions.

14. The 6502 uses the following common conventions:

- 16-bit addresses are stored with the less significant byte first. The order of the bytes is the same as in the 8080, Z-80, and 8085 microprocessors, but opposite the order used in 6800 and 6809.
- The stack pointer contains the address (on page 1) of the next available location. This convention is also used in the 6800, but the obvious alternative (last occupied location) is used in the 8080, 8085, Z-80, and 6809 microprocessors. Instructions store data in the stack using postdecrementing (they subtract 1 from the stack pointer after storing each byte) and load data from the stack using preincrementing (they add 1 to the stack pointer before loading each byte).
- The I (Interrupt) flag acts as a disable. Setting the flag (with SEI) disables the maskable interrupt and clearing the flag (with CLI) enables the maskable interrupt. This convention is the same as in the 6800 and 6809 but the opposite of that used in the 8080, 8085, and Z-80.

THE REGISTER SET

The 6502 assembly language programmer's work is complicated considerably by the processor's limited register set. In particular, there are no address-length (16-bit) user registers. Thus, variable addresses must normally be stored in pairs of memory locations on page 0 and accessed indirectly using either preindexing (indexed indirect addressing) or postindexing (indirect indexed addressing). The lack of 16-bit registers also complicates the handling of arrays or blocks that occupy more than 256 bytes of memory.

If we consider memory locations on page 0 as extensions of the register set, we may characterize the registers as follows:

- The accumulator is the center of data processing and is used as a source and destination by most arithmetic, logical, and other data processing instructions.
- Index register X is the primary index register for non-indirect uses. It is the only register that normally has a zero page indexed mode (except for the LDX STX instructions), and it is the only register that can be used for indexing with single-operand instructions such as shifts, increment, and decrement. It is also the only register that can be used for preindexing, although that mode is not common. Finally, it is the only register that can be used to load or store the stack pointer.
- Index register Y is the primary index register for indirect uses, since it is the only register that can be used for postindexing.
- Memory locations on page 0 are the only locations that can be accessed with the zero page (direct), zero page indexed, preindexed, and postindexed addressing modes.

Tables 1-1 through 1-7 contain lists of instructions having particular features. Table 1-1 lists instructions that apply only to particular registers and Table 1-2 lists instructions that can be applied directly to memory locations. Tables 1-3 through 1-7 list instructions that allow particular addressing modes: zero page (Table 1-3), absolute (Table 1-4), zero page indexed (Table 1-5), absolute indexed (Table 1-6), and preindexing and postindexing (Table 1-7).

We may describe the special features of particular registers as follows:

- **Accumulator**. Source and destination for all arithmetic and logical instructions except CPX, CPY, DEC, and INC. Only register that can be shifted with a single instruction. Only register that can be loaded or stored using preindexed or postindexed addressing.
- **Index register X**. Can be incremented using INX or decremented using DEX. Only register that can be used as an index in preindexing. Only register that can be used to load or store the stack pointer.
- **Index register Y**. Can be incremented using INY or decremented using DEY. Only register that can be used as an index in postindexing.
- **Memory locations on page 0**. Only memory locations that can hold indirect addresses for use in postindexing or preindexing. Only memory locations that can be accessed using zero page or zero page indexed addressing.
- **Status register**. Can only be stored in the stack using PHP or loaded from the stack using PLP.

Table 1-1: Registers and Applicable Instructions

Register	Instructions
A	ADC, AND, ASL, BIT, CMP, EOR, LDA, LSR, ORA, PHA, PLA, ROL, ROR, SBC, STA, TAX, TAY, TXA, TYA
P (processor status)	PHP, PLP (CLC, CLD, CLV, SEC, and SED affect particular flags)
S (stack pointer)	JSR, PHA, PHP, PLA, PLP, RTS, TSX, TXS
X	CPX, DEX, INX, LDX, STX, TAX, TSX, TXA, TXS
Y	CPY, DEY, INY, LDY, STY, TAY, TYA

Table 1-2: Instructions That Can Be Applied Directly to Memory Locations

Instruction	Function
ASL	Arithmetic shift left
BIT	Bit test (test bits 6 and 7)
DEC	Decrement by 1
INC	Increment by 1
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right

Table 1-3: Instructions That Allow Zero Page Addressing

Instruction	Function
ADC	Add with Carry
AND	Logical AND
ASL	Arithmetic shift left
BIT	Bit test
CMP	Compare memory and accumulator
CPX	Compare memory and index register X
CPY	Compare memory and index register Y
DEC	Decrement by 1
EOR	Logical EXCLUSIVE OR
INC	Increment by 1
LDA	Load accumulator
LDX	Load index register X
LDY	Load index register Y
LSR	Logical shift right
ORA	Logical OR
ROL	Rotate left
ROR	Rotate right
SBC	Subtract with Carry
STA	Store accumulator
STX	Store index register X
STY	Store index register Y

Table 1-4: Instructions That Allow Absolute (Direct) Addressing

Instruction	Function
ADC	Add with Carry
AND	Logical AND
ASL	Arithmetic shift left
BIT	Logical bit test
CMP	Compare memory and accumulator
CPX	Compare memory and index register X
CPY	Compare memory and index register Y
DEC	Decrement by 1
EOR	Logical EXCLUSIVE OR
INC	Increment by 1
JMP	Jump unconditional
JSR	Jump to subroutine
LDA	Load accumulator
LDX	Load index register X
LDY	Load index register Y
LSR	Logical shift right
ORA	Logical OR
ROL	Rotate left
ROR	Rotate right
SBC	Subtract with Carry
STA	Store accumulator
STX	Store index register X
STY	Store index register Y

Table 1-5: Instructions That Allow Zero Page Indexed Addressing

	Instruction	Function
Index Register X	ADC	Add with Carry
	AND	Logical AND
	ASL	Arithmetic shift left
	CMP	Compare memory and accumulator
	DEC	Decrement by 1
	EOR	Logical EXCLUSIVE OR
	INC	Increment by 1
	LDA	Load accumulator
	LDY	Load index register Y
	LSR	Logical shift right
	ORA	Logical OR
	ROL	Rotate left
	ROR	Rotate right
	SBC	Subtract with Carry
	STA	Store accumulator
	STY	Store index register Y
Index Register Y	LDX	Load index register X
	STX	Store index register X

Table 1-6: Instructions That Allow Absolute Indexed Addressing

	Instruction	Function
Index Register X	ADC	Add with Carry
	AND	Logical AND
	ASL	Arithmetic shift left
	CMP	Compare memory and accumulator
	DEC	Decrement by 1
	EOR	Logical EXCLUSIVE OR
	INC	Increment by 1
	LDA	Load accumulator
	LDY	Load index register Y
	LSR	Logical shift right
	ORA	Logical OR
	ROL	Rotate left
	ROR	Rotate right
	SBC	Subtract with Carry
	STA	Store accumulator
Index Register Y	ADC	Add with Carry
	AND	Logical AND
	CMP	Compare memory and accumulator
	EOR	Logical EXCLUSIVE OR
	LDA	Load accumulator
	LDX	Load index register X
	ORA	Logical OR
	SBC	Subtract with Carry
	STA	Store accumulator

Table 1-7: Instructions That Allow Postindexing and Preindexing

Instruction	Function
ADC	Add with Carry
AND	Logical AND
CMP	Compare memory and accumulator
EOR	Logical EXCLUSIVE OR
LDA	Load accumulator
ORA	Logical OR
SBC	Subtract with Carry
STA	Store accumulator

· **Stack pointer**. Always refers to an address on page 1. Can only be loaded from or stored in index register X using TXS and TSX, respectively.

Note the following:

· Almost all data processing involves the accumulator, since it provides one operand for arithmetic and logical instructions and the destination for the result.

· Only a limited number of instructions operate directly on the index registers or on memory locations. An index register can be incremented by 1, decremented by 1, or compared to a constant or to the contents of an absolute address. The data in a memory location can be incremented by 1, decremented by 1, shifted left or right, or rotated left or right.

· The available set of addressing methods varies greatly from instruction to instruction. Note in particular the limited sets available with the instructions BIT, CPX, CPY, LDX, LDY, STX, and STY.

Register Transfers

Only a limited number of direct transfers between registers are provided. A single instruction can transfer data from an index register to the accumulator, from the accumulator to an index register, from the stack pointer to index register X, or from index register X to the stack pointer. The mnemonics for the transfer instructions have the form TSD, where "S" is the source register and "D" is the destination register as in the convention proposed in IEEE Standard 694.[2] The status (P) register may only be transferred to or from the stack using PHP or PLP.

LOADING REGISTERS FROM MEMORY

The 6502 microprocessor offers many methods for loading registers from memory. The following addressing modes are available: zero page (direct), absolute (direct), immediate zero page indexed, absolute indexed, postindexed, and preindexed. Osborne[3] describes all these modes in Chapter 6 of *An Introduction to Microcomputers: Volume 1 — Basic Concepts.*

Direct Loading of Registers

The accumulator, index register X, and index register Y can be loaded from memory using direct addressing. A special zero page mode loads registers from

addresses on page 0 more rapidly than from addresses on other pages. Terminology for 6502 refers to zero page direct addressing as *zero page addressing* and to the more general direct addressing as *absolute addressing.*

Examples

1. LDA $40

This instruction loads the accumulator from memory location 0040_{16}. The special zero page addressing mode requires less time and memory than the more general absolute (direct) addressing.

2. LDX $C000

This instruction loads index register X from memory location $C000_{16}$. It uses absolute (direct) addressing.

Immediate Loading of Registers

This method can be used to load the accumulator, index register X, or index register Y with a specific value.

Examples

1. LDY #6

This instruction loads index register Y with the number 6. The 6 is an 8-bit data item, not a 16-bit address; do not confuse the number 6 with the address 0006_{16}.

2. LDA #$E3

This instruction loads the accumulator with the number $E3_{16}$.

Indexed Loading of Registers

The instructions LDA, LDX, and LDY can be used in the indexed mode. The limitations are that index register X cannot be loaded using X as an index; similarly, index register Y cannot be loaded using Y as an index. As with direct addressing, a special zero page mode is provided. Note, however, that the accumulator cannot be loaded in the zero page mode using Y as an index.

Examples

1. LDA $0340,X

This instruction loads the accumulator from the address obtained by indexing with index register X from the base address 0340_{16}; that is, the effective address is $0340_{16}+(X)$. This is the typical indexing described in *An Introduction to Microcomputers: Volume 1 — Basic Concepts.*[4]

2. LDX $40,Y

This instruction loads index register X from the address obtained by indexing with register Y from the base address 0040_{16}. Here the special zero page indexed mode saves time and memory.

Postindexed Loading of Registers

The instruction LDA can be used in the postindexed mode, in which the base address is taken from two memory locations on page 0. Otherwise, this mode is the same as regular indexing.

Example

LDA ($40),Y

This instruction loads the accumulator from the address obtained by indexing with index register Y from the base address in memory locations 0040_{16} and 0041_{16}. This mode is restricted to page 0 and index register Y. It also assumes that the indirect address is stored with its less significant byte first (at the lower address) in the usual 6502 manner.

Preindexed Loading of Registers

The instruction LDA can be used in the preindexed mode, in which the indexed address is itself used indirectly. This mode is restricted to page 0 and index register X. Note that it also assumes the existence of a table of 2-byte indirect addresses, so that only even values in X make sense.

Example

LDA ($40,X)

This instruction loads the accumulator from the indirect address obtained by indexing with register X from the base address 0040_{16}. The indirect address is in the two bytes of memory starting at $0040_{16}+(X)$. This mode is uncommon; one of its uses is to select from a table of device addresses for input/output.

Stack Loading of Registers

The instruction PLA loads the accumulator from the top of the stack and subtracts 1 from the stack pointer. The instruction PLP is similar, except that it loads the status (P) register. This is the only way to load the status register with a specific value. The index registers cannot be loaded directly from the stack, but

they can be loaded via the accumulator. The required sequences are

(for index register X)

```
PLA                 ;TOP OF STACK TO A
TAX                 ;AND ON TO X
```

(for index register Y)

```
PLA                 ;TOP OF STACK TO A
TAY                 ;AND ON TO Y
```

The stack has the following special features:

- It is always located on page 1 of memory. The stack pointer contains only the less significant byte of the next available address.
- Data is stored in the stack using postdecrementing — the instructions decrement the stack pointer by 1 *after* storing each byte. Data is loaded from the stack using preincrementing — the instructions increment the stack pointer by 1 *before* loading each byte.
- As is typical with microprocessors, there are no overflow or underflow indicators.

STORING REGISTERS IN MEMORY

The same approaches that we used to load registers from memory can also be used to store registers in memory. The only differences between loading and storing registers are

- Store instructions do not allow immediate addressing. There is no way to directly store a number in memory. Instead, it must be transferred through a register.
- STX and STY allow only zero page indexed addressing. Neither allows absolute indexed addressing.
- As you might expect, the order of operations in storing index registers in the stack is the opposite of that used in loading them from the stack. The sequences are

(for index register X)

```
TXA                 ;MOVE X TO A
PHA                 ;AND THEN TO TOP OF STACK
```

(for index register Y)

```
TYA                 ;MOVE Y TO A
PHA                 ;AND THEN TO TOP OF STACK
```

Other storage operations operate in exactly the same manner as described in the discussion of loading registers.

Examples

1. STA $50

This instruction stores the accumulator in memory location 0050_{16}. The special zero page mode is both shorter and faster than the absolute mode, since the more significant byte of the address is assumed to be 0.

2. STX $17E8

This instruction stores index register X in memory location $17E8_{16}$. It uses the absolute addressing mode with a full 16-bit address.

3. STA $A000,Y

This instruction stores the accumulator in the effective address obtained by adding index register Y to the base address $A000_{16}$. The effective address is $A000_{16}+(Y)$.

4. STA ($50),Y

This instruction stores the accumulator in the effective address obtained by adding index register Y to the base address in memory locations 0050_{16} and 0051_{16}. The instruction obtains the base address indirectly.

5. STA ($43,X)

This instruction stores the accumulator in the effective address obtained indirectly by adding index register X to the base 0043_{16}. The indirect address is in the two bytes of memory starting at $0043_{16}+(X)$.

STORING VALUES IN RAM

The normal way to initialize RAM locations is through the accumulator, one byte at a time. The programmer can also use index registers X and Y for this purpose.

Examples

1. Store an 8-bit item (VALUE) in address ADDR.

```
LDA     #VALUE      ;GET THE VALUE
STA     ADDR        ;INITIALIZE LOCATION ADDR
```

We could use either LDX, STX or LDY, STY instead of the LDA, STA sequence. Note that the 6502 treats all values the same; there is no special CLEAR instruction for generating 0s.

2. Store a 16-bit item (POINTER) in addresses ADDR and ADDR+1 (MSB in ADDR+1).

We assume that POINTER consists of POINTH (more significant byte) and POINTL (less significant byte).

```
LDA     #POINTL      ;GET LSB
STA     ADDR         ;INITIALIZE LOCATION ADDR
LDA     #POINTH      ;GET MSB
STA     ADDR+1       ;INITIALIZE LOCATION ADDR+1
```

This method allows us to initialize indirect addresses on page 0 for later use with postindexing and preindexing.

ARITHMETIC AND LOGICAL OPERATIONS

Most arithmetic and logical operations (addition, subtraction, AND, OR, and EXCLUSIVE OR) can be performed only between the accumulator and an 8-bit byte in memory. The result replaces the operand in the accumulator. Arithmetic and logical operations may use immediate, zero page (direct), absolute (direct), indexed, zero page indexed, indexed indirect, or indirect indexed addressing.

Examples

1. Add memory location 0040_{16} to the accumulator with carry.

```
ADC     $40
```

This instruction adds the contents of memory location 0040_{16} and the contents of the Carry flag to the accumulator.

2. Logically OR the accumulator with the contents of an indexed address obtained using index register X and the base $17E0_{16}$.

```
ORA     $17E0,X
```

The effective address is $17E0_{16}$+(X).

3. Logically AND the accumulator with the contents of memory location $B470_{16}$.

```
AND     $B470
```

Note the following special features of the 6502's arithmetic and logical instructions:

· The only addition instruction is ADC (Add with Carry). To exclude the Carry, you must clear it explicitly using the sequence

```
CLC                  ;MAKE CARRY ZERO
ADC     $40          ;ADD WITHOUT CARRY
```

· The only subtraction instruction is SBC (Subtract with Borrow). This instruction subtracts a memory location and the complemented Carry flag from the accumulator. SBC produces

(A) = (A) − (M) − (1−CARRY)

where M is the contents of the effective address. To exclude the Carry, you must set it explicitly using the sequence

```
SEC              ;MAKE INVERTED BORROW ONE
SBC    $40       :SUBTRACT WITHOUT CARRY
```

Note that you must set the Carry flag before a subtraction, but clear it before an addition.

· Comparison instructions perform subtractions without changing registers (except for the flags in the status register). Here we have not only CMP (Compare Memory with Accumulator), but also CPX (Compare Memory with Index Register X) and CPY (Compare Memory with Index Register Y). Note the differences between CMP and SBC; CMP does not include the Carry in the subtraction, change the accumulator, or affect the Overflow flag.

· There is no explicit Complement instruction. However, you can complement the accumulator by EXCLUSIVE ORing it with a byte which contains all 1s (11111111_2 or FF_{16}). Remember, the EXCLUSIVE OR of two bits is 1 if they are different and 0 if they are the same. Thus, EXCLUSIVE ORing with a 1 will produce a result of 0 if the other bit is 1 and 1 if the other bit is 0, the same as a logical complement (NOT instruction).

Thus we have the instruction

```
EOR    #%11111111  ;COMPLEMENT ACCUMULATOR
```

· The BIT instruction performs a logical AND but does not return a result to the accumulator. It affects only the flags. You should note that this instruction allows only direct addressing (zero page or absolute); it does not allow immediate or indexed addressing. More complex operations require several instructions; typical examples are the following:

· Add memory locations OPER1 and OPER2, place result in RESLT

```
LDA    OPER1       ;GET FIRST OPERAND
CLC                ;MAKE CARRY ZERO
ADC    OPER2       ;ADD SECOND OPERAND
STA    RESLT       ;SAVE SUM
```

Note that we must load the first operand into the accumulator and clear the Carry before adding the second operand.

• Add a constant (VALUE) to memory location OPER.

```
LDA     OPER        ;GET CURRENT VALUE
CLC                 ;MAKE CARRY ZERO
ADC     #VALUE      ;ADD VALUE
STA     OPER        ;STORE SUM BACK
```

If VALUE is 1, we can shorten this to

```
INC     OPER        ;ADD 1 TO CURRENT VALUE
```

Similarly, if VALUE is −1, we have

```
DEC     OPER        ;SUBTRACT 1 FROM CURRENT VALUE
```

BIT MANIPULATION

The programmer can set, clear, complement, or test bits by means of logical operations with appropriate masks. Shift instructions can rotate or shift the accumulator or a memory location. Chapter 7 contains additional examples of bit manipulation.

You may operate on individual bits in the accumulator as follows:

• Set them by logically ORing with 1s in the appropriate positions.

• Clear them by logically ANDing with 0s in the appropriate positions.

• Invert (complement) them by logically EXCLUSIVE ORing with 1s in the appropriate positions.

• Test them by logically ANDing with 1s in the appropriate positions.

Examples

1. Set bit 6 of the accumulator.

```
ORA     #%01000000  ;SET BIT 6 BY ORING WITH 1
```

2. Clear bit 3 of the accumulator.

```
AND     #%11110111  ;CLEAR BIT 3 BY ANDING WITH 0
```

3. Invert (complement) bit 2 of the accumulator.

```
EOR     #%00000100  ;INVERT BIT 2 BY XORING WITH 1
```

4. Test bit 5 of the accumulator. Clear the Zero flag if bit 5 is a logic 1 and set the Zero flag if bit 5 is a logic 0.

```
AND     #%00100000  ;TEST BIT 5 BY ANDING WITH 1
```

You can change more than one bit at a time by changing the masks.

5. Set bits 4 and 5 of the accumulator.

```
ORA     #%00110000  ;SET BITS 4 AND 5 BY ORING WITH 1
```

6. Invert (complement) bits 0 and 7 of the accumulator.

```
EOR     #%10000001   ;INVERT BITS 0 AND 7 BY XORING WITH 1
```

The only general way to manipulate bits in other registers or in memory is by moving the values to the accumulator.

- Set bit 4 of memory location 0040_{16}.

```
LDA     $40
ORA     #%00010000   ;SET BIT 4 BY ORING WITH 1
STA     $40
```

- Clear bit 1 of memory location $17E0_{16}$.

```
LDA     $17E0
AND     #%11111101   ;CLEAR BIT 1 BY ANDING WITH 0
STA     $17E0
```

An occasional, handy shortcut to clearing or setting bit 0 of a register or memory location is using an increment (INC, INX, or INY) to set it (if you know that it is 0) and a decrement (DEC, DEX, or DEY) to clear it (if you know that it is 1). If you do not care about the other bit positions, you can also use DEC or INC. These shortcuts are useful when you are storing a single 1-bit flag in a byte of memory.

The instruction LSR (ASL) shifts the accumulator or a memory location right (left) one position, filling the leftmost (rightmost) bit with a 0. Figures 1-1 and 1-2 describe the effects of these two instructions. The instructions ROL and ROR provide a circular shift (rotate) of the accumulator or a memory location as shown in Figures 1-3 and 1-4. Rotates operate as if the accumulator or memory location and the Carry flag formed a 9-bit circular register. You should note the following:

- Left shifts set the Carry to the value that was in bit position 7 and the Negative flag to the value that was in bit position 6.
- Right shifts set the Carry to the value that was in bit position 0.
- Rotates preserve all the bits, whereas LSR and ASL destroy the old Carry flag.
- Rotates allow you to move serial data between memory or the accumulator and the Carry flag. This is useful in performing serial I/O and in handling single bits of information such as Boolean indicators or parity.

Multibit shifts simply require the appropriate number of single-bit instructions.

Examples

1. Rotate accumulator right three positions.

```
ROR     A
ROR     A
ROR     A
```

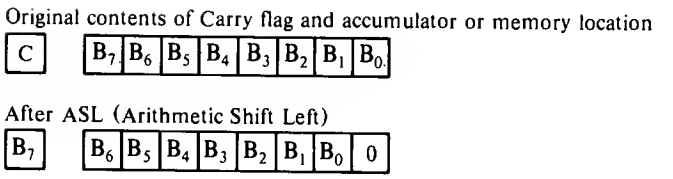


Figure 1-1: The ASL (Arithmetic Shift Left) Instruction

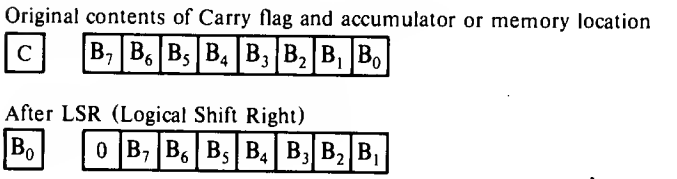


Figure 1-2: The LSR (Logical Shift Right) Instruction

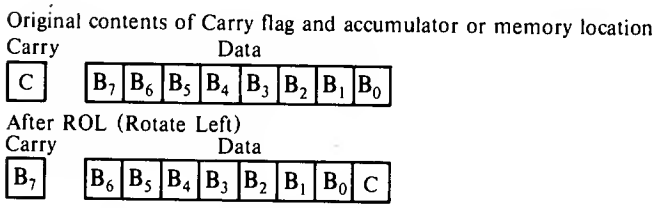


Figure 1-3: The ROL (Rotate Left) Instruction

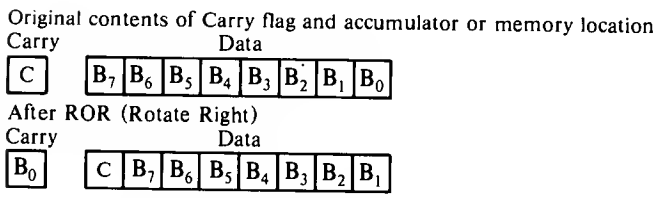


Figure 1-4: The ROR (Rotate Right) Instruction

2. Shift memory location 1700_{16} left logically four positions.

```
ASL     $1700
ASL     $1700
ASL     $1700
ASL     $1700
```

An alternative approach would be to use the accumulator; that is,

```
LDA     $1700
ASL     A
ASL     A
ASL     A
ASL     A
STA     $1700
```

The second approach is shorter (10 bytes rather than 12) and faster (16 clock cycles rather than 24), but it destroys the previous contents of the accumulator.

You can implement arithmetic shifts by using the Carry flag to preserve the current value of bit 7. Shifting right arithmetically is called *sign extension,* since it copies the sign bit to the right. A shift that operates in this manner preserves the sign of a two's complement number and can therefore be used to divide or normalize signed numbers.

Examples

1. Shift the accumulator right 1 bit arithmetically, preserving the sign (most significant) bit.

```
TAX                 ;SAVE THE ACCUMULATOR
ASL     A           ;MOVE BIT 7 TO CARRY
TXA                 ;RESTORE THE ACCUMULATOR
ROR     A           ;SHIFT THE ACCUMULATOR, COPYING BIT 7
```

When the processor performs ROR A, it moves the Carry (the old bit 7) to bit 7 and bit 7 to bit 6, thus preserving the sign of the original number.

2. Shift the accumulator left 1 bit arithmetically, preserving the sign (most significant) bit.

```
        ASL     A           ;SHIFT A, MOVING BIT 7 TO CARRY
        ROL     A           ;SAVE BIT 7 IN POSITION 0
        TAX
        LSR     A           ;CHANGE CARRY TO OLD BIT 7
        TXA
        ROR     A           ;SHIFT THE ACCUMULATOR, PRESERVING BIT 7
```

or

```
        ASL     A           ;SHIFT A, MOVING BIT 7 TO CARRY
        BCC     CLRSGN      ;WAS BIT 7 1?
        ORA     #%10000000  ;  YES, THEN KEEP IT 1
        BMI     EXIT
CLRSGN  AND     #%01111111  ;  NO, THEN KEEP IT ZERO
EXIT    NOP
```

BMI EXIT always forces a branch.

MAKING DECISIONS

We will now discuss procedures for making three types of decisions:

- Branching if a bit is set or cleared (a logic 1 or a logic 0).
- Branching if two values are equal or not equal.
- Branching if one value is greater than another or less than it.

The first type of decision allows the processor to sense the value of a flag, switch, status line, or other binary (ON/OFF) input. The second type of decision allows the processor to determine whether an input or a result has a specific value (e.g., an input is a specific character or terminator or a result is 0). The third type of decision allows the processor to determine whether a value is above or below a numerical threshold (e.g., a value is valid or invalid or is above or below a warning level or set point). Assuming that the primary value is in the accumulator and the secondary value (if needed) is in address ADDR, the procedures are as follows.

Branching Set or Cleared Bit

- Determine if a bit is set or cleared by logically ANDing the accumulator with a 1 in the appropriate bit position and 0s in the other bit positions. The Zero flag then reflects the bit value and can be used for branching (with BEQ or BNE).

Examples

1. Branch to DEST if bit 5 of the accumulator is 1.

```
AND     #%00100000  ;TEST BIT 5 OF A
BNE     DEST
```

The Zero flag is set to 1 if and only if bit 5 of the accumulator is 0. Note the inversion here.

If we assume that the data is in address ADDR, we can use the BIT instruction to produce an equivalent effect. To branch to DEST if bit 5 of ADDR is 1, we can use either

```
LDA     ADDR
AND     #%00100000
BNE     DEST
```

or

```
LDA     #%00100000
BIT     ADDR
BNE     DEST
```

We must reverse the order of the operations, since BIT does not allow immediate addressing. It does, however, leave the accumulator unchanged for later use.

2. Branch to DEST if bit 2 of the accumulator is 0.

```
AND     #%00000100  ;TEST BIT 2 OF A
BEQ     DEST
```

There are special short procedures for examining bit positions 0, 6, or 7. Bit 7 is available readily as the Negative flag after a Load or Transfer instruction; bit 0 can be moved to the Carry with LSR A or ROR A; bit 6 can be moved to the Negative flag with ASL A or ROL A.

3. Branch to DEST if bit 7 of memory location ADDR is 1.

```
LDA     ADDR        ;IS BIT 7 1?
BMI     DEST        ;YES, BRANCH
```

Note that LDA affects the Zero and Negative flags; so do transfer instructions such as TAX, TYA, TSX (but not TXS), and PLA. Store instructions (including PHA) do not affect any flags.

4. Branch to DEST if bit 6 of the accumulator is 0.

```
ASL     A           ;MOVE BIT 6 TO BIT 7
BPL     DEST
```

5. Branch to DEST if bit 0 of memory location ADDR is 1.

```
ROR     ADDR        ;MOVE BIT 0 OF ADDR TO CARRY
BCS     DEST        ;AND THEN TEST THE CARRY
```

The BIT instruction has a special feature that allows one to readily test bit 6 or bit 7 of a memory location. When the processor executes BIT, it sets the Negative flag to the value of bit 7 of the addressed memory location and the Overflow flag to the value of bit 6, regardless of the contents of the accumulator.

6. Branch to DEST if bit 7 of memory location ADDR is 0.

```
BIT     ADDR        ;TEST BIT 7 OF ADDR
BPL     DEST
```

This sequence does not affect or depend on the accumulator.

7. Branch to DEST if bit 6 of memory location ADDR is 1.

```
BIT     ADDR        ;TEST BIT 6 OF ADDR
BVS     DEST
```

This sequence requires careful documentation, since the Overflow flag is being used in a special way. Here again, the contents of the accumulator do not change or affect the sequence at all.

Branching Based on Equality

• Determine if the value in the accumulator is equal to another value by subtraction. The Zero flag will be set to 1 if the values are equal. The Compare

instruction (CMP) is more useful than the Subtract instruction (SBC) because Compare does not change the accumulator or involve the Carry.

Examples

1. Branch to DEST if the accumulator contains the number VALUE.

```
CMP     #VALUE          ;IS DATA = VALUE?
BEQ     DEST            ;YES, BRANCH
```

We could also use index register X with CPX or index register Y with CPY.

2. Branch to DEST if the contents of the accumulator are not equal to the contents of memory location ADDR.

```
CMP     ADDR            ;IS DATA = VALUE IN MEMORY?
BNE     DEST            ;NO, BRANCH
```

3. Branch to DEST if memory location ADDR contains 0.

```
LDA     ADDR            ;IS DATA ZERO?
BEQ     DEST            ;YES, BRANCH
```

We can handle some special cases without using the accumulator.

4. Branch to DEST if memory location ADDR contains 0, but do not change the accumulator or either index register.

```
INC     ADDR            ;TEST MEMORY FOR ZERO
DEC     ADDR
BEQ     DEST            :BRANCH IF IT IS FOUND
```

5. Branch to DEST if memory location ADDR does not contain 1.

```
DEC     ADDR            ;SET ZERO FLAG IF ADDR IS 1
BNE     DEST
```

This sequence, of course, changes the memory location.

6. Branch to DEST if memory location ADDR contains FF_{16}.

```
INC     ADDR            ;SET ZERO FLAG IF ADDR IS FF
BEQ     DEST
```

INC does not affect the Carry flag, but it does affect the Zero flag. Note that you cannot increment or decrement the accumulator with INC or DEC.

Branching Based on Magnitude Comparisons

· Determine if the contents of the accumulator are greater than or less than some other value by subtraction. If, as is typical, the numbers are unsigned, the Carry flag indicates which one is larger. Note that the 6502's Carry flag is a negative borrow after comparisons or subtractions, unlike the true borrow produced by such processors as the 8080, Z-80, and 6800. In general,

- Carry = 1 if the contents of the accumulator are greater than or equal to the value subtracted from it. Carry = 1 if the subtraction does not require (generate) a borrow.
- Carry = 0 if the value subtracted is larger than the contents of the accumulator. That is, Carry = 0 if the subtraction does require a borrow.

Note that the Carry is the inverse of a normal borrow. If the two operands are equal, the Carry is set to 1, just as if the accumulator were larger. If, however, you want equal values to affect the Carry as if the other value were larger, all that you must do is reverse the identities of the operands, that is, you must subtract in reverse, saving the accumulator in memory and loading it with the other value instead.

Examples

1. Branch to DEST if the contents of the accumulator are greater than or equal to the number VALUE.

```
CMP     #VALUE      ;IS DATA ABOVE VALUE?
BCS     DEST        ;YES, BRANCH
```

The Carry is set to 1 if the unsigned subtraction does not require a borrow.

2. Branch to DEST if the contents of memory address OPER1 are less than the contents of memory address OPER2.

```
LDA     OPER1       ;GET FIRST OPERAND
CMP     OPER2       ;IS IT LESS THAN SECOND OPERAND?
BCC     DEST        ;YES, BRANCH
```

The Carry will be set to 0 if the subtraction requires a borrow.

3. Branch to DEST if the contents of memory address OPER1 are less than or equal to the contents of memory address OPER2.

```
LDA     OPER2       ;GET SECOND OPERAND
CMP     OPER1       ;IS IT GREATER THAN OR EQUAL TO FIRST?
BCS     DEST        ;YES, BRANCH
```

If we loaded the accumulator with OPER1 and compared to OPER2, we could branch only on the conditions

- OPER1 greater than or equal to OPER2 (Carry set)
- OPER1 less than OPER2 (Carry cleared)

Since neither of these is what we want, we must handle the operands in the opposite order.

If the values are signed, we must allow for the possible occurrence of two's complement overflow. This is the situation in which the difference between the numbers cannot be contained in seven bits and, therefore, changes the sign of the result. For example, if one number is +7 and the other is −125, the difference is

−132, which is beyond the capacity of eight bits (it is less than −128, the most negative number that can be contained in eight bits).

Thus, in the case of signed numbers, we must allow for the following two possibilities:

- The result has the sign (positive or negative, as shown by the Negative flag) that we want, and the Overflow flag indicates that the sign is correct.
- The result does not have the sign that we want, but the Overflow flag indicates that two's complement overflow has changed the real sign.

We have to look for both a true positive (the sign we want, unaffected by overflow) or a false negative (the opposite of the sign we want, but inverted by two's complement overflow).

Examples

1. Branch to DEST if the contents of the accumulator (a signed number) are greater than or equal to the number VALUE.

```
        SEC                     ;CLEAR INVERTED BORROW
        SBC     #VALUE          ;PERFORM THE SUBTRACTION
        BVS     FNEG
        BPL     DEST            ;TRUE POSITIVE, NO OVERFLOW
        BMI     DONE
FNEG    BMI     DEST            ;FALSE NEGATIVE, OVERFLOW
DONE    NOP
```

2. Branch to DEST if the contents of the accumulator (a signed number) are less than the contents of address ADDR.

```
        SEC                     ;CLEAR INVERTED BORROW
        SBC     ADDR            ;PERFORM THE SUBTRACTION
        BVS     FNEG
        BMI     DEST            ;TRUE POSITIVE, NO OVERFLOW
        BPL     DONE
FNEG    BPL     DEST            ;FALSE NEGATIVE, OVERFLOW
DONE    NOP
```

Note that we must set the Carry and use SBC, because CMP does not affect the Overflow flag.

Tables 1-8 and 1-9 summarize the common instruction sequences used to make decisions with the 6502 microprocessor. Table 1-8 lists the sequences that depend only on the value in the accumulator; Table 1-9 lists the sequences that depend on numerical comparisons between the contents of the accumulator and a specific value or the contents of a memory location. Tables 1-10 and 1-11 contain the sequences that depend on an index register or on the contents of a memory location alone.

Table 1-8: Decision Sequences Depending on the Accumulator Alone

Condition	Flag-Setting Instruction	Conditional Branch
Any bit of A = 0	AND #MASK (1 in bit position)	BEQ
Any bit of A = 1	AND #MASK (1 in bit position)	BNE
Bit 7 of A = 0	ASL A or ROL A	BCC
	CMP #0 (preserves A)	BPL
Bit 7 of A = 1	ASL A or ROL A	BCS
	CMP #0 (preserves A)	BMI
Bit 6 of A = 0	ASL A or ROL A	BPL
Bit 6 of A = 1	ASL A or ROL A	BMI
Bit 0 of A = 0	LSR A or ROR A	BCC
Bit 0 of A = 1	LSR A or ROR A	BCS
(A) = 0	LDA, PLA, TAX, TAY, TXA, or TYA	BEQ
(A) ≠ 0	LDA, PLA, TAX, TAY, TXA, or TYA	BNE
(A) positive (MSB = 0)	LDA, PLA, TAX, TAY, TXA, or TYA	BPL
(A) negative (MSB = 1)	LDA, PLA, TAX, TAY, TXA, or TYA	BMI

Table 1-9: Decision Sequences Depending on Numerical Comparisons

Condition	Flag-Setting Instruction	Conditional Branch
(A) = VALUE	CMP #VALUE	BEQ
(A) ≠ VALUE	CMP #VALUE	BNE
(A) ≥ VALUE (unsigned)	CMP #VALUE	BCS
(A) < VALUE (unsigned)	CMP #VALUE	BCC
(A) = (ADDR)	CMP ADDR	BEQ
(A) ≠ (ADDR)	CMP ADDR	BNE
(A) ≥ (ADDR) (unsigned)	CMP ADDR	BCS
(A) < (ADDR) (unsigned)	CMP ADDR	BCC

Table 1-10: Decision Sequences Depending on an Index Register

Condition	Flag-Setting Instruction	Conditional Branch
(X or Y) = VALUE	CPX or CPY #VALUE	BEQ
(X or Y) ≠ VALUE	CPX or CPY #VALUE	BNE
(X or Y) ≥ VALUE (unsigned)	CPX or CPY #VALUE	BCS
(X or Y) < VALUE (unsigned)	CPX or CPY #VALUE	BCC
(X or Y) = (ADDR)	CPX or CPY ADDR	BEQ
(X or Y) ≠ (ADDR)	CPX or CPY ADDR	BNE
(X or Y) ≥ (ADDR) (unsigned)	CPX or CPY ADDR	BCS
(X or Y) < (ADDR) (unsigned)	CPX or CPY ADDR	BCC

Table 1-11: Decision Sequences Depending on a Memory Location Alone

Condition	Flag-Setting Instruction(s)	Conditional Branch
Bit 7 = 0	BIT ADDR	BPL
	ASL ADDR or ROL ADDR	BCC
Bit 7 = 1	BIT ADDR	BMI
	ASL ADDR or ROL ADDR	BCS
Bit 6 = 0	BIT ADDR	BVC
	ASL ADDR or ROL ADDR	PBL
Bit 6 = 1	BIT ADDR	BVS
	ASL ADDR or ROL ADDR	BMI
(ADDR) = 0	INC ADDR, DEC ADDR	BEQ
(ADDR) ≠ 0	INC ADDR, DEC ADDR	BNE
Bit 0 = 0	LSR ADDR or ROR ADDR	BCC
Bit 0 = 1	LSR ADDR or ROR ADDR	BCS

LOOPING

The simplest way to implement a loop (that is, repeat a sequence of instructions) with the 6502 microprocessor is as follows:

1. Load an index register or memory location with the number of times the sequence is to be executed.
2. Execute the sequence.
3. Decrement the index register or memory location by 1.
4. Return to Step 2 if the result of Step 3 is not 0.

Typical programs look like this:

```
        LDX     #NTIMES       ;COUNT = NUMBER OF REPETITIONS
LOOP    .
        .
        .  instructions to be repeated
        .
        DEX
        BNE     LOOP
```

Nothing except clarity stops us from counting up (using INX, INY, or INC); of course, you must change the initialization appropriately. As we will see later, a 16-bit counter is much easier to increment than it is to decrement. In any case, the instructions to be repeated must not interfere with the counting of the repetitions. You can store the counter in either index register or any memory location. Index register X's special features are its use in preindexing and the wide availability of zero page indexed modes. Index register Y's special feature is its use in postindexing. As usual, memory locations on page 0 are shorter and faster to use than are memory locations on other pages.

Of course, if you use an index register or a single memory location as a counter, you are limited to 256 repetitions. You can provide larger numbers of repetitions by nesting loops that use a single register or memory location or by using a pair of memory locations as illustrated in the following examples:

- Nested loops

```
        LDX     #NTIMM        ;START OUTER COUNTER
LOOPO   LDY     #NTIML        ;START INNER COUNTER
LOOPI   .
        .
        .  instructions to be repeated
        .
        DEY                   ;DECREMENT INNER COUNTER
        BNE     LOOPI
        DEX                   ;DECREMENT OUTER COUNTER
        BNE     LOOPO
```

The outer loop restores the inner counter (index register Y) to its starting value

(NTIML) after each decrement of the outer counter (index register X). The nesting produces a multiplicative factor — the instructions starting at LOOPI are repeated NTIMM × NTIML times. Of course, a more general (and more reasonable) approach would use two memory locations on page 0 instead of two index registers.

- 16-bit counter in two memory locations

```
        LDA     #NTIMLC         ;INITIALIZE LSB OF COUNTER
        STA     COUNTL
        LDA     #NTIMHC         ;INITIALIZE MSB OF COUNTER
        STA     COUNTH
LOOP    .
        .   instructions to be repeated
        .
        INC     NTIMLC          ;INCREMENT LSB OF COUNTER
        BNE     LOOP
        INC     NTIMHC          ;AND CARRY TO MSB OF COUNTER IF NEEDED
        BNE     LOOP
```

The idea here is to increment only the less significant byte unless there is a carry to the more significant byte. Note that we can recognize a carry only by checking the Zero flag, since INC does not affect the Carry flag. Counting up is much simpler than counting down; the comparable sequence for decrementing a 16-bit counter is

```
        LDA     NTIML           ;IS LSB OF COUNTER ZERO?
        BNE     CNTLSB
        DEC     NTIMH           ;YES, BORROW FROM MSB
CNTLSB  DEC     NTIML           ;DECREMENT LSB OF COUNTER
        BNE     LOOP            ;CONTINUE IF LSB HAS NOT REACHED ZERO
        LDA     NTIMH           ;OR IF MSB HAS NOT REACHED ZERO
        BNE     LOOP
```

If we count up, however, we must remember to initialize the counter to the complement of the desired value (indicated by the names NTIMLC and NTIMHC in the program using INC).

ARRAY MANIPULATION

The simplest way to access a particular element of an array is by using indexed addressing. One can then

1. Manipulate the element by indexing from the starting address of the array.

2. Access the succeeding element (at the next higher address) by incrementing the index register using INX or INY, or access the preceding element (at the next lower address) by decrementing the index register using DEX or DEY. One

could also change the base; this is simple if the base is an absolute address, but awkward if it is an indirect address.

3. Access an arbitrary element by loading an index register with its index. Typical array manipulation procedures are easy to program if the array is one-dimensional, the elements each occupy 1 byte, and the number of elements is less than 256. Some examples are

• Add an element of an array to the accumulator. The base address of the array is a constant BASE. Update index register X so that it refers to the succeeding 8-bit element.

```
        ADC     BASE,X          ;ADD CURRENT ELEMENT
        INX                     ;ADDRESS NEXT ELEMENT
```

• Check to see if an element of an array is 0 and add 1 to memory location ZCOUNT if it is. Assume that the address of the array is a constant BASE and its index is in index register X. Update index register X so that it refers to the preceding 8-bit element.

```
        LDA     BASE,X          ;GET CURRENT ELEMENT
        BNE     UPDDT           ;IS ITS VALUE ZERO?
        INC     ZCOUNT          ;YES, ADD 1 TO COUNT OF ZEROS
UPDDT   DEX                     ;ADDRESS PRECEDING ELEMENT
```

• Load the accumulator with the 35th element of an array. Assume that the starting address of the array is BASE.

```
        LDX     #35             ;GET INDEX OF REQUIRED ELEMENT
        LDA     BASE,X          ;OBTAIN THE ELEMENT
```

The most efficient way to process an array is to start at the highest address and work backward. This is the best approach because it allows you to count the index register down to 0 and exit when the Zero flag is set. You must adjust the initialization and the indexed operations slightly to account for the fact that the 0 index is never used. The changes are

• Load the index register with the number of elements.

• Use the base address START−1, where START is the lowest address actually occupied by the array.

If, for example, we want to perform a summation starting at address START and continuing through LENGTH elements, we use the program

```
        LDX     #LENGTH         ;START AT THE END OF THE ARRAY
        LDA     #0              ;CLEAR THE SUM INITIALLY
ADBYTE  CLC
        ADC     START-1,X       ;ADD THE NEXT ELEMENT
        DEX
        BNE     ADBYTE          ;COUNT ELEMENTS
```

Manipulating array elements becomes more difficult if you need more than one element during each iteration (as in a sort that requires interchanging of elements), if the elements are more than one byte long, or if the elements are themselves addresses (as in a table of starting addresses). The basic problem is the lack of 16-bit registers or 16-bit instructions. The processor can never be instructed to handle more than 8 bits. Some examples of more general array manipulation are

· Load memory locations POINTH and POINTL with a 16-bit element of an array (stored LSB first). The base address of the array is BASE and the index of the element is in index register X. Update X so that it points to the next 16-bit element.

```
        LDA     BASE,X      ;GET LSB OF ELEMENT
        STA     POINTL
        INX
        LDA     BASE,X      ;GET MSB OF ELEMENT
        STA     POINTH
        INX                 ;ADDRESS NEXT ELEMENT
```

The single instruction LDA BASE+1,X loads the accumulator from the same address as the sequence

```
        INX
        LDA     BASE,X
```

assuming that X did not originally contain FF_{16}. If, however, we are using a base address indirectly, the alternatives are

```
        INC     PGZRO       ;INCREMENT BASE ADDRESS
        BNE     INDEX
        INC     PGZRO+1     ;WITH CARRY IF NECESSARY
INDEX   LDA     (PGZRO),Y
```

or

```
        INY
        LDA     (PGZRO),Y
```

The second sequence is much shorter, but the first sequence will handle arrays that are more than 256 bytes long.

· Exchange an element of an array with its successor if the two are not already in descending order. Assume that the elements are 8-bit unsigned numbers. The base address of the array is BASE and the index of the first number is in index register X.

```
        LDA     BASE,X      ;GET ELEMENT
        CMP     BASE+1,X    ;IS SUCCESSOR SMALLER?
        BCS     DONE        ;NO, NO INTERCHANGE NECESSARY
        PHA                 ;YES, SAVE ELEMENT
        LDA     BASE+1,X    :INTERCHANGE
        STA     BASE,X
        PLA
        STA     BASE+1,X
DONE    INX                 ;ACCESS NEXT ELEMENT
```

• Load the accumulator from the 12th indirect address in a table. Assume that the table starts at the address BASE.

```
        LDX     #24             ;GET DOUBLED OFFSET FOR INDEX
        LDA     BASE,X          ;GET LSB OF ADDRESS
        STA     PGZRO           ;SAVE ON PAGE ZERO
        INX
        LDA     BASE,X          ;GET MSB OF ADDRESS
        STA     PGZRO+1         ;SAVE ON PAGE ZERO
        LDY     #0
        LDA     (PGZRO),Y       ;LOAD INDIRECT BY INDEXING WITH ZERO
```

Note that you must double the index to handle tables containing addresses, since each 16-bit address occupies two bytes of memory.

If the entire table is on page 0, we can use the preindexed (indexed indirect) addressing mode.

```
        LDX     #24             ;GET DOUBLED OFFSET FOR INDEX
        LDA     (BASE,X)        ;LOAD FROM INDEXED INDIRECT ADDRESS
```

You still must remember to double the index. Here we must also initialize the table of indirect addresses in the RAM on page 0.

We can generalize array processing by storing the base address in two locations on page 0 and using the postindexed (indirect indexed) addressing mode. Now the base address can be a variable. This mode assumes the use of page 0 and index register Y and is available only for a limited set of instructions.

Examples

1. Add an element of an array to the accumulator. The base address of the array is in memory locations PGZRO and PGZRO+1. The index of the element is in index register Y. Update index register Y so that it refers to the succeeding 8-bit element.

```
CLC     ADC     (PGZRO),Y       ;ADD CURRENT ELEMENT
        INY                     ;ADDRESS NEXT ELEMENT
```

2. Check to see if an element of an array is 0 and add 1 to memory location ZCOUNT if it is. Assume that the base address of the array is in memory locations PGZRO and PGZRO+1. The index of the element is in index register Y. Update index register Y so that it refers to the preceding 8-bit element.

```
        LDA     (PGZRO),Y       ;GET CURRENT ELEMENT
        BNE     UPDDT           ;IS ITS VALUE ZERO?
        INC     ZCOUNT          ;YES, ADD 1 TO COUNT OF ZEROS
UPDDT   DEY                     ;ADDRESS PRECEDING ELEMENT
```

Postindexing also lets us handle arrays that occupy more than 256 bytes. As we noted earlier, the simplest approach to long counts is to keep a 16-bit complemented count in two memory locations. If the array is described by a base address on page 0, we can update that base whenever we update the more significant byte of the complemented count. For example, if we want to clear an area of memory

described by a complemented count in memory locations COUNTH and COUNTL and an initial base address in memory locations PGZRO and PGZRO+1, we can use the following program:

```
        LDA     #0              ;DATA = ZERO
        TAY                     ;INDEX = ZERO
CLEAR   STA     (PGZRO),Y       ;CLEAR A BYTE
        INY                     ;MOVE TO NEXT BYTE
        BNE     CHKCNT
        INC     PGZRO + 1       ;AND TO NEXT PAGE IF NEEDED
CHKCNT  INC     COUNTL          ;COUNT BYTES
        BNE     CLEAR
        INC     COUNTH          ;WITH CARRY TO MSB
        BNE     CLEAR
```

The idea here is to proceed to the next page by incrementing the more significant byte of the indirect address when we finish a 256-byte section.

One can also simplify array processing by reducing the multiplications required in indexing to additions. In particular, one can handle arrays of two-byte elements by using ASL A to double an index in the accumulator.

Example

Load the accumulator from the indirect address indexed by the contents of memory location INDEX. Assume that the table starts at address BASE.

```
        LDA     INDEX           ;GET INDEX
        ASL     A               ;AND DOUBLE IT FOR 2-BYTE ENTRIES
        TAX
        LDA     BASE,X          ;GET LSB OF INDIRECT ADDRESS
        STA     PGZRO           ;SAVE ON PAGE ZERO
        INX
        LDA     BASE,X          ;GET MSB OF INDIRECT ADDRESS
        STA     PGZRO + 1       ;SAVE ON PAGE ZERO
        LDY     #0              ;PREINDEX WITH ZERO
        LDA     (PGZRO),Y
```

As before, if the entire table of indirect addresses is on page 0, we can use the preindexed (indexed indirect) addressing mode.

```
        LDA     INDEX           ;GET INDEX
        ASL     A               ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        TAX
        LDA     (BASE,X)        ;LOAD FROM INDEXED INDIRECT ADDRESS
```

You can handle indexing into longer arrays by using the postindexed (indirect indexed) mode. Here we must construct a base address with an explicit addition before indexing, since the 6502's index registers are only 8 bits long.

Example

Load the accumulator from the element of an array defined by a starting address BASE (BASEH more significant byte, BASEL less significant byte) and a 16-bit index in memory locations INDEX and INDEX+1 (MSB in INDEX+1).

```
LDA     #BASEL          ;MOVE LSB OF BASE TO PAGE ZERO
STA     PGZRO
LDA     #BASEH          ;ADD MSB'S OF BASE AND INDEX
STA     POINTL
CLC
ADC     INDEX+1
STA     PGZRO+1
LDY     INDEX           ;USE LSB OF INDEX EXPLICITLY
LDA     (PGZRO),Y       ;GET ELEMENT
```

TABLE LOOKUP

Table lookup can be handled by the same procedures as array manipulation. Some examples are

• Load the accumulator with an element from a table. Assume that the base address of the table is BASE (a constant) and the 8-bit index is in memory location INDEX.

```
LDX     INDEX           ;GET INDEX
LDA     BASE,X          ;GET THE ELEMENT
```

The problem is more complicated if INDEX is a 16-bit number.

• Load the accumulator with an element from a table. Assume that the base address of the table is BASE (a constant, made up of bytes BASEH and BASEL) and the 16-bit index is in memory locations INDEX and INDEX+1 (MSB in INDEX+1).

The procedure is the same one we just showed for an array. You must add the more significant byte of the index to the more significant byte of the base with an explicit addition. You can then use postindexing to obtain the element.

• Load memory locations POINTH and POINTL with a 16-bit element from a table. Assume that the base address of the table is BASE (a constant) and the index is in memory location INDEX.

```
LDA     INDEX           ;GET THE INDEX
ASL     A               ;DOUBLE IT FOR TWO-BYTE ENTRIES
TAX
LDA     BASE,X          ;GET LSB OF ELEMENT
INX
LDA     BASE,X          ;GET MSB OF ELEMENT
STA     POINTH
```

We can also handle the case in which the base address is a variable in two memory locations on page 0 (PGZRO and PGZRO+1).

```
        LDA     INDEX           ;GET THE INDEX
        ASL     A               ;DOUBLE IT FOR TWO-BYTE ENTRIES
        TAY
        LDA     (PGZRO),Y       ;GET LSB OF ELEMENT
        STA     POINTL
        INY
        LDA     (PGZRO),Y       ;GET MSB OF ELEMENT
        STA     POINTH
```

We can revise the program further to handle an array with more than 128 entries.

```
        LDA     INDEX           ;GET THE INDEX
        ASL     A               ;DOUBLE IT FOR TWO-BYTE ENTRIES
        BCC     LDELEM
        INC     PGZRO+1         ;ADD CARRY TO INDIRECT ADDRESS
LDELEM  TAY
        LDA     (PGZRO),Y       ;GET LSB OF ELEMENT
        STA     POINTL
        INY
        LDA     (PGZRO),Y       ;GET MSB OF ELEMENT
        STA     POINTH
```

Still another extension handles a 16-bit index.

```
        LDA     INDEX           ;GET LSB OF INDEX
        ASL     A               ;DOUBLE IT
        TAY
        LDA     INDEX+1         ;GET MSB OF INDEX
        ROL     A               ;DOUBLE IT WITH CARRY
        ADC     PGZRO+1         ;AND ADD RESULT TO INDIRECT ADDRESS
        STA     PGZRO+1
        LDA     (PGZRO),Y       ;GET LSB OF ELEMENT
        STA     POINTL
        INY
        LDA     (PGZRO),Y       ;GET MSB OF ELEMENT
        STA     POINTH
```

· Transfer control (jump) to a 16-bit address obtained from a table. Assume that the base address of the table is BASE (a constant) and the index is in memory location INDEX.

Here there are two options: Store the address obtained from the table in two memory locations and use an indirect jump, or store the address obtained from the table in the stack and use the RTS (Return from Subroutine) instruction.

OPTION 1: Indirect Jump

```
        LDA     INDEX           ;GET INDEX
        ASL     A               ;DOUBLE IT FOR TWO-BYTE ENTRIES
        TAX
        LDA     BASE,X          ;GET LSB OF DESTINATION ADDRESS
        STA     TEMP            ;STORE LSB SOMEWHERE
        INX
```

```
LDA     BASE,X      ;GET MSB OF DESTINATION ADDRESS
STA     TEMP+1      ;STORE MSB IN NEXT BYTE
JMP     (TEMP)      ;INDIRECT JUMP TO DESTINATION
```

JMP is the only 6502 instruction that has true indirect addressing. Note that TEMP and TEMP+1 can be anywhere in memory; they need not be on page 0.

OPTION 2: Jump Through the Stack

```
LDA     INDEX       ;GET INDEX
ASL     A           ;DOUBLE IT FOR TWO-BYTE ENTRIES
TAX
INX
LDA     BASE,X      :GET MSB OF DESTINATION ADDRESS
PHA                 ;SAVE MSB IN STACK
DEX
LDA     BASE,X      ;GET LSB OF DESTINATION ADDRESS
PHA                 ;SAVE LSB IN STACK
RTS                 ;TRANSFER CONTROL TO DESTINATION
```

This alternative is awkward for the following reasons:

- RTS adds 1 to the program counter after loading it from the stack. Thus, the addresses in the table must all be one less than the actual values to which you wish to transfer control. This offset evidently speeds the processor's execution of the JSR (Jump to Subroutine) instruction, but it also can confuse the programmer.

- You must remember that the stack is growing down in memory, toward lower addresses. To have the destination address end up in its normal order (less significant byte at lower address), we must push the more significant byte first. This is essentially a double negative; we store the address in the wrong order but it ends up right because the stack is growing down.

- The use of RTS is confusing. How can one *return* from a routine that one has never called? In fact, this approach uses RTS to call a subroutine. You should remember that RTS is simply a jump instruction that obtains the new value for the program counter from the top of the stack. While the common use of RTS is to transfer control from a subroutine back to a main program (hence, the mnemonic), there is no reason to limit it to that function. The mnemonic may confuse the programmer, but the microprocessor does exactly what it is supposed to do. Careful documentation can help calm the nerves if you feel uneasy about this procedure.

The common uses of jump tables are to implement CASE statements (for example, multiway branches as used in languages such as FORTRAN, Pascal,

and PL/I) to decode commands from a keyboard, and to respond to function keys on a terminal.

CHARACTER MANIPULATION

The easiest way to manipulate characters is to treat them as unsigned 8-bit numbers. The letters and digits form ordered subsequences of the ASCII characters; for example, the ASCII representation of the letter A is one less than the ASCII representation of the letter B. Handling one character at a time is just like handling normal 8-bit unsigned numbers. Some examples are

- Branch to address DEST if the accumulator contains an ASCII E.

```
        CMP     #'E         ;IS DATA E?
        BEQ     DEST        ;YES, BRANCH
```

- Search a string starting at address STRNG until a non-blank character is found.

```
        LDX     #0          ;POINT TO START OF STRING
        LDA     #'          ;GET A BLANK FOR CHECKING
EXAMC   CMP     STRNG,X     ;IS NEXT CHARACTER A BLANK?
        BNE     DONE        ;NO, DONE
        INX                 ;YES,PROCEED TO NEXT CHARACTER
        JMP     EXAMC
DONE    NOP
```

or

```
        LDX     #$FF        ;POINT TO BYTE BEFORE START
        LDA     #'          ;GET A BLANK FOR COMPARISON
EXAMC   INX                 ;PROCEED TO NEXT CHARACTER
        CMP     STRNG,X     ;IS IT A BLANK?
        BEQ     EXAMC       ;YES, KEEP LOOKING
```

- Branch to address DEST if the accumulator contains a letter between C and F, inclusive.

```
        CMP     #'C         ;IS DATA BELOW C?
        BCC     DONE        ;YES, DONE
        CMP     #'G         ;IS DATA BELOW G?
        BCC     DEST        ;YES, MUST BE BETWEEN C AND F
DONE    NOP
```

Chapter 8 contains further examples of string manipulation.

CODE CONVERSION

You can convert data from one code to another using arithmetic or logical operations (if the relationship is simple) or lookup tables (if the relationship is complex).

Examples

1. Convert an ASCII digit to its binary-coded decimal (BCD) equivalent.

```
SEC                 ;CLEAR THE INVERTED BORROW
SBC     #'0         ;CONVERT ASCII TO BCD
```

Since the ASCII digits form an ordered subsequence, all you must do is subtract the offset (ASCII 0).

You can also clear bit positions 4 and 5 with the single instruction

```
AND     #%11001111  :CONVERT ASCII TO BCD
```

Either the arithmetic sequence or the logical instruction will, for example, convert ASCII 0 (30_{16}) to decimal 0 (00_{16}).

2. Convert a binary-coded decimal (BCD) digit to its ASCII equivalent.

```
CLC                 ;CLEAR THE CARRY
ADC     #'0         ;CONVERT BCD TO ASCII
```

The inverse conversion is equally simple. You can also set bit positions 4 and 5 with the single instruction

```
ORA     #%00110000  ;CONVERT BCD TO ASCII
```

Either the arithmetic sequence or the logical instruction will, for example, convert decimal 6 (06_{16}) to ASCII 6 (36_{16}).

3. Convert one 8-bit code to another using a lookup table. Assume that the lookup table starts at address NEWCD and is indexed by the value in the original code (for example, the 27th entry is the value in the new code corresponding to 27 in the original code). Assume that the data is in memory location CODE.

```
LDX     CODE        ;GET THE OLD CODE
LDA     NEWCD,X     ;CONVERT IT TO THE NEW CODE
```

Chapter 4 contains further examples of code conversion.

MULTIPLE-PRECISION ARITHMETIC

Multiple-precision arithmetic requires a series of 8-bit operations. One must

- Clear the Carry before starting addition or set the Carry before starting subtraction, since there is never a carry into or borrow from the least significant byte.
- Use the Add with Carry (ADC) or Subtract with Borrow (SBC) instruction to perform an 8-bit operation and include the carry or borrow from the previous operation.

A typical 64-bit addition program is

```
        LDX     #8              ;NUMBER OF BYTES = 8
        CLC                     ;CLEAR CARRY TO START
ADDB:   LDA     NUM1-1,X        ;GET A BYTE OF ONE OPERAND
        ADC     NUM2-1,X        ;ADD A BYTE OF THE OTHER OPERAND
        STA     NUM1-1,X        ;STORE THE 8-BIT SUM
        DEX
        BNE     ADDB            ;COUNT BYTE OPERATIONS
```

Chapter 6 contains further examples.

MULTIPLICATION AND DIVISION

Multiplication can be implemented in a variety of ways. One technique is to convert simple multiplications to additions or left shifts.

Examples

1. Multiply the contents of the accumulator by 2.

```
        ASL     A               ;DOUBLE A
```

2. Multiply the contents of the accumulator by 5.

```
        STA     TEMP
        ASL     A               ;A TIMES 2
        ASL     A               ;A TIMES 4
        ADC     TEMP            ;A TIMES 5
```

This approach assumes that shifting the accumulator left never produces a carry. This approach is often handy in determining the locations of elements of two-dimensional arrays. For example, let us assume that we have a set of temperature readings taken at four different positions in each of three different tanks. We organize the readings as a two-dimensional array T(I,J), where I is the tank number (1, 2, or 3) and J is the number of the position in the tank (1, 2, 3, or 4). We store the readings in the linear memory of the computer one after another as follows, starting with tank 1:

```
BASE            T(1,1)      Reading at tank 1, location 1
BASE+1          T(1,2)      Reading at tank 1, location 2
BASE+2          T(1,3)      Reading at tank 1, location 3
BASE+3          T(1,4)      Reading at tank 1, location 4
BASE+4          T(2,1)      Reading at tank 2, location 1
BASE+5          T(2,2)      Reading at tank 2, location 2
BASE+6          T(2,3)      Reading at tank 2, location 3
BASE+7          T(2,4)      Reading at tank 2, location 4
BASE+8          T(3,1)      Reading at tank 3, location 1
BASE+9          T(3,2)      Reading at tank 3, location 2
BASE+10         T(3,3)      Reading at tank 3, location 3
BASE+11         T(3,4)      Reading at tank 3, location 4
```

So, generally the reading T(I,J) is located at address BASE+4 × (I−1) + (J−1). If I is in memory location IND1 and J is in memory location IND2, we can load the accumulator with T(I,J) as follows:

```
LDA     IND1        ;GET I
SEC
SBC     #1          ;CALCULATE I - 1
ASL     A           ;2 X (I - 1)
ASL     A           ;4 X (I - 1)
SEC
SBC     #1          ;4 X (I - 1) - 1
CLC
ADC     IND2        ;4 X (I - 1) + J - 1
TAX
LDA     BASE,X      ;GET T(I,J)
```

We can extend this approach to handle arrays with more dimensions.

Obviously, the program is much simpler if we store I−1 in memory location IND1 and J−1 in memory location IND2. We can then load the accumulator with T(I,J) using

```
LDA     IND1        ;GET I - 1
ASL     A           ;2 X (I - 1)
ASL     A           ;4 X (I - 1)
CLC
ADC     IND2        ;4 X (I - 1) + (J - 1)
TAX
LDA     BASE,X      ;GET T(I.J)
```

- Simple divisions can also be implemented as right logical shifts.

Example

Divide the contents of the accumulator by 4.

```
LSR     A           ;DIVIDE BY 2
LSR     A           ;AND BY 2 AGAIN
```

If you are multiplying or dividing signed numbers, you must be careful to separate the signs from the magnitudes. You must replace logical shifts with arithmetic shifts that preserve the value of the sign bit.

- Algorithms involving shifts and additions (multiplication) or shifts and subtractions (division) can be used as described in Chapter 6.

- Lookup tables can be used as discussed previously in this chapter.

Chapter 6 contains additional examples of arithmetic programs.

LIST PROCESSING[5]

Lists can be processed like arrays if the elements are stored in consecutive addresses. If the elements are queued or chained, however, the limitations of the instruction set are evident in that

- No 16-bit registers or instructions are available.
- Indirect addressing is allowed only through pointers on page 0.
- No true indirect addressing is available except for JMP instructions.

Examples

1. Retrieve an address stored starting at the address in memory locations PGZRO and PGZRO+1. Place the retrieved address in memory locations POINTL and POINTH.

```
        LDY     #0              ;INDEX = ZERO
        LDA     (PGZRO),Y       ;GET LSB OF ADDRESS
        STA     POINTL
        INY
        LDA     (PGZRO),Y       ;GET MSB OF ADDRESS
        STA     POINTH
```

This procedure allows you to move from one element to another in a linked list.

2. Retrieve data from the address currently in memory locations PGZRO and PGZRO+1 and increment that address by 1.

```
        LDY     #0              ;INDEX = ZERO
        LDA     (PGZRO),Y       ;GET DATA USING POINTER
        INC     PGZRO           ;UPDATE POINTER BY 1
        BNE     DONE
        INC     PGZRO+1
DONE    NOP
```

This procedure allows you to use the address in memory as a pointer to the next available location in a buffer. Of course, you can also leave the pointer fixed and increment a buffer index. If that index is in memory location BUFIND, we have

```
        LDY     BUFIND          ;GET BUFFER INDEX
        LDA     (PGZRO),Y       ;GET DATA FROM BUFFER
        INC     BUFIND          :UPDATE BUFFER INDEX BY 1
```

3. Store an address starting at the address currently in memory locations PGZRO and PGZRO+1. Increment the address in memory locations PGZRO and PGZRO+1 by 2.

```
        LDY     #0              ;INDEX = ZERO
        LDA     #ADDRL          ;SAVE LSB OF ADDRESS
        STA     (PGZRO),Y
        LDA     #ADDRH          ;SAVE MSB OF ADDRESS
        INY
        STA     (PGZRO),Y
        CLC                     ;INCREMENT POINTER BY 2
        LDA     PGZRO
        ADC     #2
        STA     PGZRO
        BCC     DONE            ;WITH CARRY IF NECESSARY
        INC     PGZRO+1
DONE    NOP
```

This procedure lets you build a list of addresses. Such a list could be used, for example, to write threaded code in which each routine concludes by transferring control to its successor. The list could also contain the starting addresses of a series of test procedures or tasks or the addresses of memory locations or I/O devices assigned by the operator to particular functions. Of course, some lists may have to be placed on page 0 in order to use the 6502's preindexed or postindexed addressing modes.

GENERAL DATA STRUCTURES[6]

More general data structures can be processed using the procedures that we have described for array manipulation, table lookup, and list processing. The key limitations in the instruction set are the same ones that we mentioned in the discussion of list processing.

Examples

1. **Queues or linked lists**. Assume that we have a queue header consisting of the address of the first element in memory locations HEAD and HEAD+1 (on page 0). If there are no elements in the queue, HEAD and HEAD+1 both contain 0. The first two locations in each element contain the address of the next element or 0 if there is no next element.

· Add the element in memory locations PGZRO and PGZRO+1 to the head of the queue.

```
LDX     PGZRO          ;REPLACE HEAD, SAVING OLD VALUE
LDA     HEAD
STX     HEAD
PHA
LDA     PGZRO+1
LDX     HEAD+1
STA     HEAD+1
LDY     #0             ;INDEX = ZERO
PLA                    ;NEW HEAD'S LINK IS OLD HEAD
STA     (HEAD),Y
TXA
INY
STA     (HEAD),Y
```

· Remove an element from the head of the queue and set the Zero flag if no element is available.

```
LDY     #0             ;GET ADDRESS OF FIRST ELEMENT
LDA     (HEAD),Y       ;GET LESS SIGNIFICANT BYTE
STA     PGZRO
INY
LDA     (HEAD),Y       ;GET MORE SIGNIFICANT BYTE
```

```
        STA     PGZRO+1
        ORA     PGZRO           ;ANY ELEMENTS IN QUEUE?
        BEQ     DONE            ;NO, DONE (LINK = 0000)
        LDA     (PGZRO),Y       ;YES, MAKE NEXT ELEMENT NEW HEAD
        STA     (HEAD),Y
        DEY
        LDA     (PGZRO),Y
        STA     (HEAD),Y
        INY                     ;CLEAR ZERO FLAG BY MAKING Y 1
DONE    NOP
```

Note that we can use the sequence

```
        LDA     ADDR
        ORA     ADDR+1
```

to test the 16-bit number in memory locations ADDR and ADDR+1. The Zero flag is set only if both bytes are 0.

2. **Stacks**. Assume that we have a stack structure consisting of 8-bit elements. The address of the next empty location is in addresses SPTR and SPTR+1 on page 0. The lowest address that the stack can occupy is LOW and the highest address is HIGH.

· If the stack overflows, clear the Carry flag and exit. Otherwise, store the accumulator in the stack and increase the stack pointer by 1. Overflow means that the stack has exceeded its area.

```
        LDA     #HIGHL          :STACK POINTER GREATER THAN HIGH?
        CMP     SPTR
        LDA     #HIGHM
        SBC     SPTR+1
        BCC     EXIT            ;YES, CLEAR CARRY AND EXIT (OVERFLOW)
        LDY     #0              ;NO STORE ACCUMULATOR IN STACK
        STA     (SPTR),Y
        INC     SPTR            ;INCREMENT STACK POINTER
        BNE     EXIT
        INC     SPTR+1
EXIT    NOP
```

· If the stack underflows, set the Carry flag and exit. Otherwise, decrease the stack pointer by 1 and load the accumulator from the stack. Underflow means that there is nothing left in the stack.

```
        LDA     #LOWL           ;STACK POINTER AT OR BELOW LOW?
        CMP     SPTR
        LDA     #LOWM
        SBC     SPTR+1
        BCS     EXIT            ;YES, SET CARRY AND EXIT (UNDERFLOW)
        LDA     SPTR            ;NO, DECREMENT STACK POINTER
        BNE     NOBOR
        DEC     SPTR+1
NOBOR   DEC     SPTR
        LDY     #0              :LOAD ACCUMULATOR FROM STACK
        LDA     (SPTR),Y
EXIT    NOP
```

PARAMETER PASSING TECHNIQUES

The most common ways to pass parameters on the 6502 microprocessor are

1. **In registers**. Three 8-bit registers are available (A, X, and Y). This approach is adequate in simple cases but it lacks generality and can handle only a limited number of parameters. The programmer must remember the normal uses of the registers in assigning parameters. In other words,

- The accumulator is the obvious place to put a single 8-bit parameter.
- Index register X is the obvious place to put an index, since it is the most accessible and has the most instructions that use it for addressing. Index register X is also used in preindexing (indexed indirect addressing).
- Index register Y is used in postindexing (indirect indexed addressing).

This approach is reentrant as long as the interrupt service routines save and restore all the registers.

2. **In an assigned area of memory**. The easiest way to implement this approach is to place the starting address of the assigned area in two memory locations on page 0. The calling routine must store the parameters in memory and load the starting address into the two locations on page 0 before transferring control to the subroutine. This approach is general and can handle any number of parameters, but it requires a large amount of management. If you assign different areas of memory for each call or each routine, you are essentially creating your own stack. If you use a common area of memory, you lose reentrancy. In this method, the programmer is responsible for assigning areas of memory, avoiding interference between routines, and saving and restoring the pointers required to resume routines after subroutine calls or interrupts. The extra memory locations on page 0 must be treated like registers.

3. **In program memory immediately following the subroutine call**. If you use this approach, you must remember the following:

- The starting address of the memory area minus 1 is at the top of the stack. That is, the starting address is the normal return address, which is 1 larger than the address the 6502's JSR instruction saves in the stack. You can move the starting address to memory locations RETADR and RETADR+1 on page 0 with the following sequence:

```
        PLA                     ;GET LSB OF RETURN ADDRESS
        STA     RETADR
        PLA                     ;GET MSB OF RETURN ADDRESS
        STA     RETADR+1
        INC     RETADR          ;ADD 1 TO RETURN ADDRESS
        BNE     DONE
        INC     RETADR+1
DONE    NOP
```

Now we can access the parameters through the indirect address. That is, you can load the accumulator with the first parameter by using the sequence

```
LDY     #0              ;INDEX = ZERO
LDA     (RETADR),Y      ;LOAD FIRST PARAMETER
```

An obvious alternative is to leave the return address unchanged and start the index at 1. That is, we would have

```
PLA                     ;GET LSB OF RETURN ADDRESS
STA     RETADR
PLA                     ;GET MSB OF RETURN ADDRESS
STA     RETADR+1
```

Now we could load the accumulator with the first parameter by using the sequence

```
LDY     #1              ;INDEX = 1
LDA     (RETADR),Y      ;LOAD FIRST PARAMETER
```

- All parameters must be fixed for a given call, since the program memory is typically ROM.

- The subroutine must calculate the actual return address (the address of the last byte in the parameter area) and place it on top of the stack before executing a Return from Subroutine (RTS) instruction.

Example

Assume that subroutine SUBR requires an 8-bit parameter and a 16-bit parameter. Show a main program that calls SUBR and contains the required parameters. Also show the initial part of the subroutine that retrieves the parameters, storing the 8-bit item in the accumulator and the 16-bit item in memory locations PGZRO and PGZRO+1, and places the correct return address at the top of the stack.

Subroutine call

```
        JSR     SUBR            ;EXECUTE SUBROUTINE
        .BYTE   PAR8            ;8-BIT PARAMETER
        .WORD   PAR16           ;16-BIT PARAMETER
        ... next instruction ...
```

Subroutine

```
SUBR    PLA                     ;GET LSB OF PARAMETER ADDRESS
        STA     RETADR
        PLA                     ;GET MSB OF PARAMETER ADDRESS
        STA     RETADR+1
        LDY     #1              ;ACCESS FIRST PARAMETER
        LDA     (RETADR),Y      ;GET FIRST PARAMETER
        TAX
        INY
        LDA     (RETADR),Y      ;ACCESS LSB OF 16-BIT PARAMETER
        STA     PGZRO
        INY
        LDA     (RETADR),Y      ;GET MSB OF 16-BIT PARAMETER
```

```
        STA     PGZRO+1
        LDA     RETADR      ;CALCULATE ACTUAL RETURN ADDRESS
        CLC
        ADA     #3
        TAY
        BCC     STRMSB
        INC     RETADR+1
STRMSB  LDA     RETADR+1    ;PUT RETURN ADDRESS ON TOP OF STACK
        PHA
        TYA
        PHA
```

The initial sequence pops the return address from the top of the stack (JSR saved it there) and stores it in memory locations RETADR and RETADR+1. In fact, the return address does not contain an instruction; instead, it contains the first parameter. Remember that JSR actually saves the return address minus 1; that is why we must start the index at 1 rather than at 0. Finally, adding 3 to the return address and saving the sum in the stack lets a final RTS instruction transfer control back to the instruction following the parameters.

This approach allows parameters lists of any length. However, obtaining the parameters from memory and adjusting the return address is awkward at best; it becomes a longer and slower process as the number of parameters increases.

4. **In the stack**. If you use this approach, you must remember the following:

- JSR stores the return address at the top of the stack. The parameters that the calling routine placed in the stack begin at address 01ss + 3, where ss is the contents of the stack pointer. The 16-bit return address occupies the top two locations and the stack pointer itself always refers to the next empty address, not the last occupied one. Before the subroutine can obtain its parameters, it must remove the return address from the stack and save it somewhere.

- The only way for the subroutine to determine the value of the stack pointer is by using the instruction TSX. After TSX has been executed, you can access the top of the stack by indexing with register X from the base address 0101_{16}. The extra offset of 1 is necessary because the top of the stack is empty.

- The calling program must place the parameters in the stack before calling the subroutine.

- Dynamically allocating space on the stack is difficult at best. If you wish to reduce the stack pointer by NRESLT, two general approaches are

```
        TSX                 ;MOVE STACK POINTER TO A VIA X
        TXA
        SEC                 ;SUBTRACT NRESLT FROM POINTER
        SBC     #NRESLT
        TAX                 ;RETURN DIFFERENCE TO STACK POINTER
        TXS
```

or

```
        LDX     #NRESLT         ;COUNT = NRESLT
PUSHB   PHA                     ;MOVE STACK POINTER DOWN 1
        DEX
        BNE     PUSHB
```

Either approach leaves NRESLT empty locations at the top of the stack as shown in Figure 1-5. Of course, if NRESLT is 1 or 2, simply executing PHA the appropriate number of times will be much faster and shorter. The same approaches can be used to provide stack locations for temporary storage.

Example

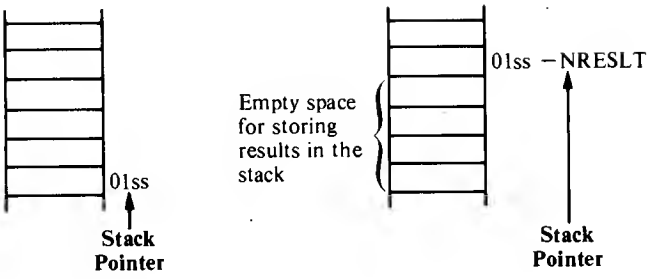
Assume that subroutine SUBR requires an 8-bit parameter and a 16-bit parameter, and that it produces two 8-bit results. Show a call of SUBR, the removal of the return address from the stack, and the cleaning of the stack after the return. Figure 1-6 shows the appearance of the stack initially, after the subroutine call, and at the end. If you always use the stack for parameters and results, you will generally keep the parameters at the top of the stack in the proper order. Then you will not have to save the parameters or assign space in the stack for the results (they will replace some or all of the original parameters). You will, however, have to assign space on the stack for temporary storage to maintain generality and reentrancy.

Calling program

```
        TSX                     ;LEAVE ROOM ON STACK FOR RESULTS
        TXA                     ;A GENERAL WAY TO ADJUST SP
        CLC
        ADC     #2
        TAX
        TXS
        LDA     #PAR16H         ;MOVE 16-BIT PARAMETER TO STACK
        PHA
        LDA     #PAR16L
        PHA
        LDA     #PAR8           ;MOVE 8-BIT PARAMETER TO STACK
        PHA
        JSR     SUBR            ;EXECUTE SUBROUTINE
        TSX                     ;CLEAN PARAMETERS FROM STACK
        TXA
        CLC
        ADC     #3
        TAX
        TXS                     ;RESULT IS NOW AT TOP OF STACK
```

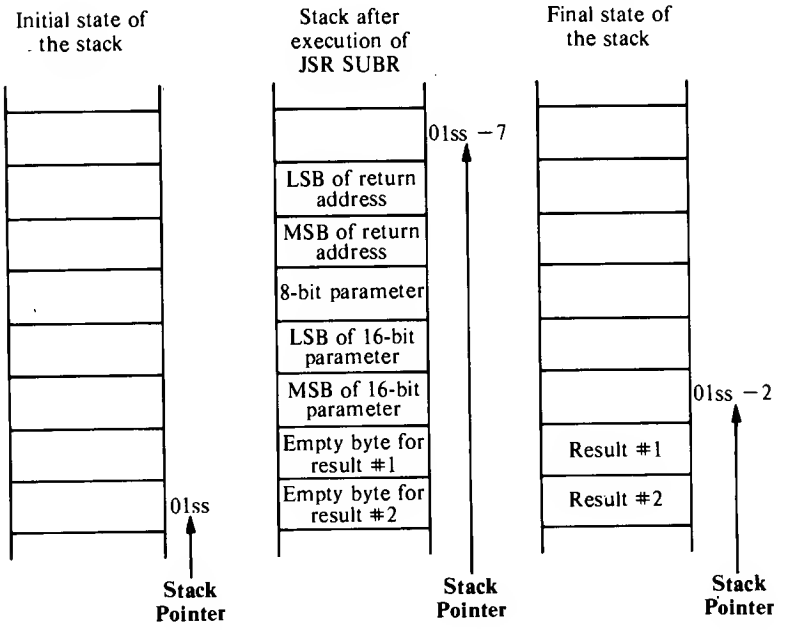
Subroutine

```
SUBR    PLA                     ;REMOVE RETURN ADDRESS FROM STACK
        STA     RETADR
        PLA
        STA     RETADR+1
```



No values are placed in the locations.
The initial contents of the stack pointer are ss.

Figure 1-5: The Stack Before and After Assigning NRESLT Empty Locations for Results



The initial contents of the stack pointer are ss.

Figure 1-6: The Effect of a Subroutine on the Stack

SIMPLE INPUT/OUTPUT

Simple input/output can be performed using any memory addresses and any instructions that reference memory. The most common instructions are the following:

- LDA (load accumulator) transfers eight bits of data from an input port to the accumulator.
- STA (store accumulator) transfers eight bits of data from the accumulator to an output port.

Other instructions can combine the input operation with arithmetic or logical operations. Typical examples are the following:

- AND logically ANDs the contents of the accumulator with the data from an input port.
- BIT logically ANDs the contents of the accumulator with the data from an input port but does not store the result anywhere. It does, however, load the Negative flag with bit 7 of the input port and the Overflow flag with bit 6, regardless of the contents of the accumulator.
- CMP subtracts the data at an input port from the contents of the accumulator, setting the flags but leaving the accumulator unchanged.

Instructions that operate on data in memory can also be used for input or output. Since these instructions both read and write memory, their effect on input and output ports may be difficult to determine. Remember, input ports cannot generally be written, nor can output ports generally be read. The commonly used instructions are the following:

- ASL shifts its data to the left, thus moving bit 7 to the Carry for possible serial input.
- DEC decrements its data. Among other effects, this inverts bit 0.
- INC increments its data. Among other effects, this inverts bit 0.
- LSR shifts its data to the right, thus moving bit 0 to the Carry for possible serial input.
- ROR rotates its data to the right, thus moving the old Carry to bit 7 and moving bit 0 to the Carry.
- ROL rotates its data to the left, thus moving the old Carry to bit 0 and moving bit 7 to the Carry.

The effects of these instructions on an input port are typically similar to their effects on a ROM location. The microprocessor can read the data, operate on it, and set the flags, but it cannot store the result back into memory. The effects on

an output port are even stranger, unless the port is latched and buffered. If it is not, the data that the processor reads is system-dependent and typically has no connection with what was last stored there.

Examples

1. Perform an 8-bit input operation from the input port assigned to memory address $B000_{16}$.

```
LDA     $B000          ;INPUT DATA
```

2. Perform an 8-bit output operation to the output port assigned to memory address $3A5E_{16}$.

```
STA     $3A5E          ;OUTPUT DATA
```

3. Set the Zero flag if bit 5 of the input port assigned to memory address $75D0_{16}$ is 0.

```
LDA     #%00100000     ;GET MASK
AND     $75D0          ;SET FLAG IF BIT 5 IS ZERO
```

We can also use the sequence

```
LDA     #%00100000     ;GET MASK
BIT     $75D0          ;SET FLAG IF BIT 5 IS ZERO
```

If the bit position of interest is number 6, we can use the single instruction

```
BIT     $75D0
```

to set the Overflow flag to its value.

4. Set the Zero flag if the data at the input port assigned to memory address 1700_{16} is $1B_{16}$.

```
LDA     #$1B
CMP     $1700
```

5. Load the Carry flag with the data from bit 7 of the input port assigned to memory address $33A5_{16}$.

```
ASL     $33A5
```

Note that this instruction does not change the data in memory location $33A5_{16}$ unless that location is latched and buffered. If, for example, there are eight simple switches attached directly to the port, the instruction will surely have no effect on whether the switches are open or closed.

6. Place a logic 1 in bit 0 of the output port assigned to memory address $B070_{16}$.

```
LDA     $B070
ORA     #%00000001
STA     $B070
```

If none of the other bits in address $B070_{16}$ are connected, we can use the sequence

```
SEC
ROL     $B070
```

If we know that bit 0 of address $B070_{16}$ is currently a logic 0, we can use the single instruction

```
INC     $B070
```

All of these alternatives will have strange effects if memory address $B070_{16}$ cannot be read. The first two will surely make bit 0 a logic 1, but their effects on the other bits are uncertain. The outcome of the third alternative would be a total mystery, since we would have no idea what is being incremented. We can avoid the uncertainty by saving a copy of the data in RAM location TEMP. Now we can operate on the copy using

```
LDA     TEMP          ;GET COPY OF OUTPUT DATA
ORA     #%00000001    ;SET BIT 0
STA     $B070         ;OUTPUT NEW DATA
STA     TEMP          ;AND SAVE A COPY OF IT
```

LOGICAL AND PHYSICAL DEVICES

One way to select I/O devices by number is to use an I/O device table. An I/O device table assigns the actual I/O addresses (*physical devices*) to the device numbers (*logical devices*) to which a program refers. Using this method, a program written in a high-level language may refer to device number 2 for input and number 5 for output. For testing purposes, the operator may assign devices numbers 2 and 5 to be the input and output ports, respectively, of his or her terminal. For normal stand-alone operation, the operator may assign device number 2 to be an analog input unit and device number 5 the system printer. If the system is to be operated by remote control, the operator may assign devices numbers 2 and 5 to be communications units used for input and output.

One way to provide this distinction between logical and physical devices is to use the 6502's indexed indirect addressing or preindexing. This mode assumes that the device table is located on page 0 and is accessed via an index in register X. If we have a device number in memory location DEVNO, the following programs can be used:

- Load the accumulator from the device number given by the contents of memory location DEVNO.

```
LDA     DEVNO         ;GET DEVICE NUMBER
ASL     A             ;DOUBLE IT TO HANDLE DEVICE ADDRESSES
TAX
LDA     (DEVTAB,X)    ;GET DATA FROM DEVICE
```

- Store the accumulator in the device number given by the contents of memory location DEVNO.

```
PHA                     ;SAVE THE DATA
LDA     DEVNO           ;GET DEVICE NUMBER
ASL     A               ;DOUBLE IT TO HANDLE DEVICE ADDRESSES
TAX
PLA
STA     (DEVTAB,X)      ;SEND DATA TO DEVICE
```

In both cases, we assume that the I/O device table starts at address DEVTAB (on page 0) and consists of 2-byte addresses. Note that the 6502 provides an appropriate addressing method, but does not produce any error messages if the programmer uses that method improperly by accessing odd addresses or by indexing off the end of page 0 (the processor does provide automatic wraparound). In real applications (see Chapter 10), the device table will probably contain the starting addresses of I/O subroutines (*drivers*) rather than actual device addresses.

STATUS AND CONTROL

You can handle status and control signals like any other data. The only special problem is that the processor may not be able to read output ports; in that case, you must retain copies (in RAM) of the data sent to those ports.

Examples

1. Branch to address DEST if bit 3 of the input port assigned to memory address $A100_{16}$ is 1.

```
LDA     $A100           ;GET INPUT DATA
AND     #%00001000      ;MASK OFF BIT 3
BNE     DEST
```

2. Branch to address DEST if bits 4, 5, and 6 of the input port assigned to address STAT are 5 (101 binary).

```
LDA     STAT            ;GET STATUS
AND     #%01110000      ;MASK OFF BITS 4, 5, AND 6
CMP     #%01010000      ;IS STATUS FIELD 5?
BEQ     DEST            ;YES, BRANCH
```

3. Set bit 5 of address CNTL to 1.

```
LDA     CNTL            ;GET CURRENT DATA FROM PORT
ORA     #%00100000      ;SET BIT 5
STA     CNTL            ;RESTORE DATA TO PORT
```

If address CNTL cannot be read properly, we can use a copy in memory address TEMP.

```
LDA     TEMP            ;GET CURRENT DATA FROM PORT
ORA     #%00100000      ;SET BIT 5
STA     CNTL            ;RESTORE DATA TO PORT
STA     TEMP            ;UPDATE COPY OF DATA
```

You must update the copy every time you change the data.

4. Set bits 2, 3, and 4 of address CNTL to 6 (110 binary).

```
LDA     CNTL          ;GET CURRENT DATA FROM PORT
AND     #%11100011    ;CLEAR BITS 2, 3, AND 4
ORA     #%00011000    ;SET CONTROL FIELD TO 6
STA     CNTL          ;RESTORE DATA TO PORT
```

As in example 3, if address CNTL cannot be read properly, we can use a copy in memory address TEMP.

```
LDA     TEMP          ;GET CURRENT DATA FROM PORT
AND     #%11100011    ;CLEAR BITS 2, 3, AND 4
ORA     #%00011000    ;SET CONTROL FIELD TO 6
STA     CNTL          ;UPDATE PORT
STA     TEMP          ;UPDATE COPY OF DATA
```

Retaining copies of the data in memory (or using the values stored in a latched, buffered output port) allows you to change part of the data without affecting other parts that may have unrelated meanings. For example, you could change the state of one indicator light (for example, a light that indicated local or remote operation) without affecting other indicator lights attached to the same port. You could similarly change one control line (for example, a line that determined whether motion was in the positive or negative X-direction) without affecting other control lines attached to the same port.

PERIPHERAL CHIPS

The major peripheral chips in 6502-based microcomputers are the 6520 and 6522 parallel interfaces (known as the Peripheral Interface Adapter or PIA and the Versatile Interface Adapter or VIA, respectively), the 6551 and 6850 serial interfaces (known as Asynchronous Communications Interface Adapters or ACIAs), and the 6530 and 6532 multifunction devices (known as ROM-I/O-timers or RAM-I/O-timers or ROM-RAM-I/O-timers, abbreviated RIOT or RRIOT and sometimes called *combo* chips). All of these devices can perform a variety of functions, much like the microprocessor itself. Of course, peripheral chips perform fewer different functions than processors and the range of functions is limited. The idea behind programmable peripheral chips is that each contains many useful circuits; the designer selects the ones he or she wants to use by storing one or more selection codes in control registers, much as one selects a particular circuit from a Designer's Casebook by turning to a particular page. The advantages of programmable chips are that a single board containing such devices can handle many applications and changes, or, corrections can be made by changing selection codes rather than by redesigning circuit boards. The disadvantages

of programmable chips are the lack of standards and the difficulty of determining how specific chips operate.

Chapter 10 contains typical initialization routines for the 6520, 6522, 6551, 6850, 6530, and 6532 devices. These devices are also discussed in detail in the *Osborne 4 and 8-Bit Microprocessor Handbook*[7]. We will provide only a brief overview of the 6522 device here, since it is the most widely used. 6522 devices are used, for example, in the Rockwell AIM-65, Synertek SYM-1, Apple, and other popular microcomputers as well as in add-on I/O boards and other functions available from many manufacturers.

6522 Parallel Interface (Versatile Interface Adapter)

A VIA contains two 8-bit parallel I/O ports (A and B), four status and control lines (CA1, CA2, CB1, and CB2 — two for each of the two ports), two 16-bit counter/timers (timer 1 and timer 2), an 8-bit shift register, and interrupt logic. Each VIA occupies 16 memory addresses. The RS (register select) lines choose the various internal registers as described in Table 1-12. The way that a VIA operates is determined by the values that the program stores in four registers.

- Data Direction Register A (DDRA) determines whether the pins on port A are inputs (0s) or outputs (1s). A data direction register determines only the direction in which traffic flows; you may compare it to a directional arrow that indicates which way traffic can move on a highway lane or railroad track. The data direction register does not affect what data flows or how often it changes; it only affects the direction. Each pin in the I/O port has a corresponding bit in the data direction register, and thus, each pin can be selected individually as either an input or an output. Of course, the most common choices are to make an entire 8-bit I/O port input or outport by storing 0s or 1s in all eight bits of the data direction register.

- Data Direction Register B (DDRB) similarly determines whether the pins in port B are inputs or outputs.

- The Peripheral Control Register (PCR) determines how the handshaking control lines (CA1, CB1, CA2, and CB2) operate. Figure 1-7 contains the bit assignments for this register. We will discuss briefly the purposes of these bits and their uses in common applications.

- The Auxiliary Control Register (ACR) determines whether the input data ports are latched and how the timers and shift register operate. Figure 1-8 contains the bit assignments for this register. We will also discuss briefly the purposes of these bits and their uses in common applications.

Table 1-12: Addressing the Internal Registers of the 6522 VIA

Label	Select Lines				Addressed Location
	RS3	RS2	RS1	RS0	
DEV	0	0	0	0	Output register for I/O Port B
DEV+1	0	0	0	1	Output register for I/O Port A, with handshaking
DEV+2	0	0	1	0	I/O Port B Data Direction register
DEV+3	0	0	1	1	I/O Port A Data Direction register
DEV+4	0	1	0	0	Read Timer 1 Counter low-order byte Write to Timer 1 Latch low-order byte
DEV+5	0	1	0	1	Read Timer 1 Counter high-order byte Write to Timer 1 Latch high-order byte and initiate count
DEV+6	0	1	1	0	Access Timer 1 Latch low-order byte
DEV+7	0	1	1	1	Access Timer 1 Latch high-order byte
DEV+8	1	0	0	0	Read low-order byte of Timer 2 and reset Counter interrupt Write to low-order byte of Timer 2 but do not reset interrupt
DEV+9	1	0	0	1	Access high-order byte of Timer 2, reset Counter interrupt on write
DEV+A	1	0	1	0	Serial I/O Shift register
DEV+B	1	0	1	1	Auxiliary Control register
DEV+C	1	1	0	0	Peripheral Control register
DEV+D	1	1	0	1	Interrupt Flag register
DEV+E	1	1	1	0	Interrupt Enable register
DEV+F	1	1	1	1	Output register for I/O Port A, without handshaking

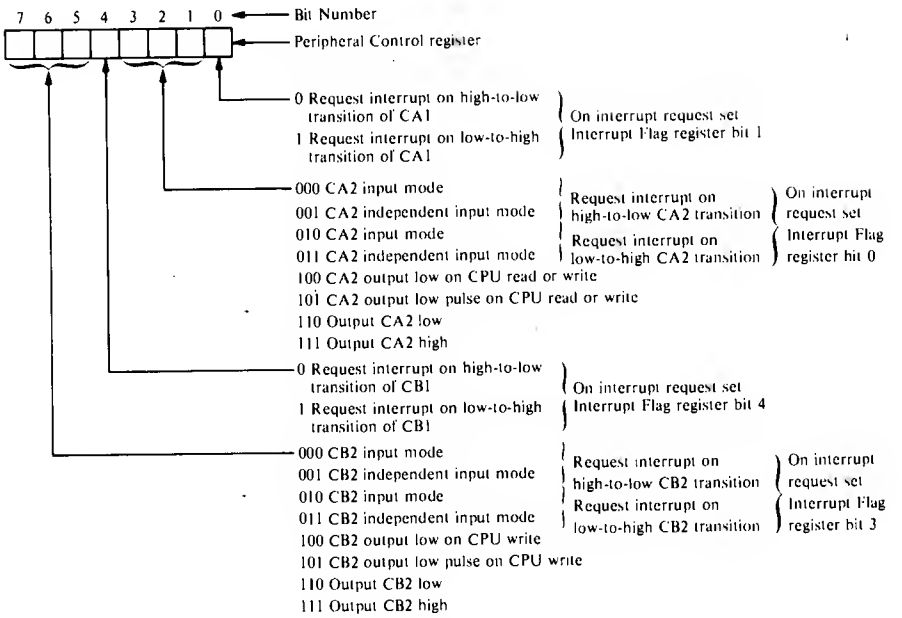


Figure 1-7: Bit Assignments for the 6522 VIA's Peripheral Control Register

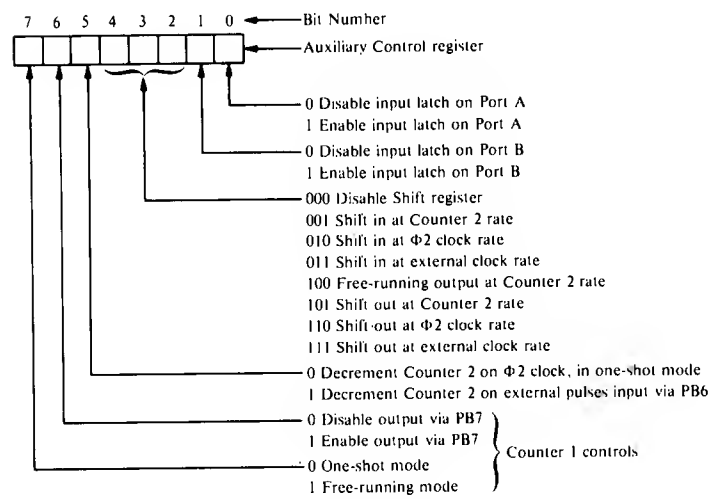


Figure 1-8: Bit Assignments for the 6522 VIA's Auxiliary Control Register

In order to initialize a VIA properly, we must know what its start-up state is. Reset clears all the VIA registers, thus making all the data and control lines inputs, disabling all latches, interrupts, and other functions, and clearing all status bits.

The data direction registers are easy to initialize. Typical routines are

- Make port A input.

```
LDA     #0
STA     DDRA
```

- Make port B output.

```
LDA     #%11111111
STA     DDRB
```

- Make bits 0 through 3 of port A input, bits 4 through 7 output.

```
LDA     #$11110000
STA     DDRA
```

- Make bit 0 of port B input, bits 1 through 7 output.

```
LDA     #$11111110
STA     DDRB
```

Bit 0 could, for example, be a serial input line.

The Peripheral Control Register is more difficult to initialize. We will briefly discuss the purposes of the control lines before showing some examples.

Control lines CA1, CA2, CB1, and CB2 are basically intended for use as handshaking signals. In a handshake, the sender indicates the availability of data by means of a transition on a serial line; the transition tells the receiver that new data is available to it on the parallel lines. Common names for this serial line are VALID DATA, DATA READY, and DATA AVAILABLE. In response to this signal, the receiver must accept the data and indicate its acceptance by means of a transition on another serial line. This transition tells the sender that the latest parallel data has been accepted and that another transmission sequence can begin. Common names for the other serial line are DATA ACCEPTED, PERIPHERAL READY, BUFFER EMPTY, and DATA ACKNOWLEDGE.

Typical examples of complete sequences are

- **Input from a keyboard**. When the operator presses a key, the keyboard produces a parallel code corresponding to the key and a transition on the DATA READY or VALID DATA line. The computer must determine that the transition has occurred, read the data, and produce a transition on the DATA ACCEPTED line to indicate that the data has been read.

- **Output to a printer**. The printer indicates to the computer that it is ready by means of a transition on a BUFFER EMPTY or PERIPHERAL READY line. Note that PERIPHERAL READY is simply the inverse of DATA ACCEPTED, since the peripheral obviously cannot be ready as long as it has not accepted the

latest data. The computer must determine that the printer is ready, send the data, and produce a transition on the DATA READY line to indicate that new data is available. Of course, input and output are in some sense mirror images. In the input case, the peripheral is the sender and the computer is the receiver; in the output case, the computer is the sender and the peripheral is the receiver.

Thus, a chip intended for handshaking functions must provide the following functions:

- It must recognize the appropriate transitions on the DATA READY or PERIPHERAL READY lines.
- It must provide an indication that the transition has occurred in a form that is easy for the computer to handle.
- It must allow for the production of the response — that is, for the computer to indicate DATA ACCEPTED to an input peripheral or DATA READY to an output peripheral.

There are some obvious variations that the handshaking chip should allow for, including the following:

- The active transition may be either a high-to-low edge (a trailing edge) or a low-to-high edge (a leading edge). If the chip does not allow either one, we will need extra inverter gates in some situations, since both polarities are common.
- The response may require either a high-to-low edge or a low-to-high edge. In fact, it may require either a brief pulse or a long signal that lasts until the peripheral begins its next operation.

Experience has shown that the handshaking chip can provide still more convenience, at virtually no cost, in the following ways:

- It can latch the transitions on the DATA READY or PERIPHERAL READY lines, so that they are held until the computer is ready for them. The computer need not monitor the status lines continuously to avoid missing a transition.
- It can clear the status latches automatically when an input port is read or an output port is written, thus preparing them for the next operation.
- It can produce the response automatically when an input port is read or an output port is written, thus eliminating the need for additional instructions. This option is known as an *automatic mode.* The problem with any automatic mode, no matter how flexible the designers make it, is that it will never satisfy all applications. Thus, most chips also provide a mode in which the program retains control over the response; this mode is called a *manual mode.*
- In cases where the peripherals are simple switches or lights and do not need

any status or control signals, the chip should allow independent operation of the status lines. The designer can then use these lines (which would otherwise be wasted) for such purposes as threshold detection, zero-crossing detection, or clock inputs. In such cases, the designer wants the status and control signals to be entirely independent of the operations on the parallel port. We should not have any automatic clearing of latches or sending of responses. This is known as an *independent mode.*

The 6522 peripheral control register allows the programmer to provide any of these functions. Bits 0 through 3 govern the operation of port B and its control signals; bits 4 through 7 govern the operation of port A and its control signals. The status indicators are in the Interrupt flag register (see Figure 1-9). We may characterize the bits in the control register as follows:

- Bit 0 (for port A) and bit 4 (for port B) determine whether the active transition on control line 1 is high-to-low (0) or low-to-high (1). If control line 2 is an extra input, bit 2 (for port A) and bit 6 (for port B) has a similar function.

- If control line 2 is an extra input, bit 1 (for port A) and bit 5 (for port B) determine whether it operates independently of the parallel data port. This bit is 0 for normal handshaking and 1 for independent operation.

- Bit 3 (for port A) and bit 7 (for port B) determine whether control line 2 is an extra input line (0) or an output response (1).

- If control line 2 is an output response, bit 2 (for port A) and bit 6 (for port B) determine whether it operates in an automatic mode (0) or a manual mode (1).

- If control line 2 is being operated in the automatic mode, bit 1 (for port A) and bit 5 (for port B) determine whether the response lasts for one clock cycle (1) or until the peripheral produces another active transition on control line 1 (0).

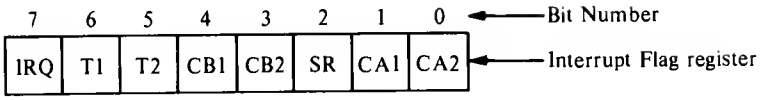
- If control line 2 is being operated in the manual mode, bit 1 (for port A) and bit 5 (for port B) determine its level.

Some typical examples are

- The peripheral indicates DATA READY or PERIPHERAL READY with a high-to-low transition on control line 1. No response is necessary.

In the 4 bits controlling a particular port, the only requirement is that bit 0 must be 0 to allow recognition of a high-to-low transition on control line 1. The other bits are arbitrary, although our preference is to clear unused bits as a standard convention. Thus, the bits would be 0000.

- The peripheral indicates DATA READY or PERIPHERAL READY with a low-to-high transition on control line 1. No response is necessary. Bit 0 must be set to 1; the other bits are arbitrary. Bit 0 determines which edge the VIA recognizes.



Bit Number	Set by	Cleared by
0	Active transition of the signal on the CA2 pin.	Reading or writing the A Port Output register (ORA) using address 0001.
1	Active transition of the signal on the CA1 pin.	Reading or writing the A Port Output register (ORA) using address 0001.
2	Completion of eight shifts.	Reading or writing the Shift register.
3	Active transition of the signal on the CB2 pin.	Reading or writing the B Port Output register.
4	Active transition of the signal on the CB1 pin.	Reading or writing the B Port Output register.
5	Time-out of Timer 2.	Reading T2 low-order counter or writing T2 high-order counter.
6	Time-out of Timer 1.	Reading T1 low-order counter or writing T1 high-order latch.
7	Active and enabled interrupt condition.	Action which clears interrupt condition.
Bits 0, 1, 3, and 4 are the I/O handshake signals. Bit 7 (IRQ) is 1 if any of the interrupts is both active and enabled.		

Figure 1-9: The 6522 VIA's Interrupt Flag Register

· The peripheral indicates DATA READY or PERIPHERAL READY with a high-to-low transition on control line 1. The port must respond by producing a pulse on control line 2 that lasts one clock cycle after the processor reads the input or writes the output.

The required 4-bit sequence is

Bit 3 = 1 to make control line 2 an output
Bit 2 = 0 to operate control line 2 in the automatic mode.

The port therefore produces the response without processor intervention.

Bit 1 = 1 to make the response last one clock cycle.
Bit 0 = 0 to recognize a high-to-low transition on control line 1.

· The peripheral indicates DATA READY or PERIPHERAL READY with a low-to-high transition on control line 1. The port must respond by bringing control line 2 low until the peripheral becomes ready again.

The required 4-bit sequence is

Bit 3 = 1 to make control line 2 an output.
Bit 2 = 0 to operate control line 2 in the automatic mode.
Bit 1 = 0 to make the response last until the peripheral becomes ready again.
Bit 0 = 1 to recognize a low-to-high transition on control line 1 as the ready signal.

· The peripheral indicates DATA READY or PERIPHERAL READY with a low-to-high transition on control line 1. The processor must respond under program control.

The required 4-bit sequence is

Bit 3 = 1 to make control line 2 an output.
Bit 2 = 1 to operate control line 2 in the manual mode.
Bit 1 is the initial state for control line.
Bit 0 = 1 to recognize a low-to-high transition on control line 1 as the ready signal.

The following sequences can be used to produce the response

```
Make CA2 a logic 1:
        LDA     VIAPCR        ;READ THE PERIPHERAL REGISTER
        ORA     #%00000010    ;SET CONTROL LINE 2 TO 1
        STA     VIAPCR
Make CA2 a logic 0:
        LDA     VIAPCR        ;READ THE PERIPHERAL REGISTER
        AND     #%11111101    ;SET CONTROL LINE 2 TO 0
        STA     VIAPCR
Make CB2 a logic 1:
        LDA     VIAPCR        ;READ THE PERIPHERAL REGISTER
        ORA     #%00100000    ;SET CONTROL LINE 2 TO 1
        STA     VIAPCR
Make CB2 a logic 0:
        LDA     VIAPCR        ;READ THE PERIPHERAL REGISTER
        AND     #%11011111    ;SET CONTROL LINE 2 TO 0
        STA     VIAPCR
```

These sequences do not depend on the contents of the peripheral control register, since they do not change any of the bits except the one that controls the response.

Tables 1-13 and 1-14 summarize the operating modes for control lines CA2 and CB2. Note that the automatic output modes differ slightly in that port A produces a response after either read or write operations, whereas port B produces a response only after write operations.

Table 1-13: Operating Modes for Control Line CA2 of a 6522 VIA

PCR3	PCR2	PCR1	Mode
0	0	0	Input Mode — Set CA2 Interrupt flag (IFR0) on a negative transition of the input signal. Clear IFR0 on a read or write of the Peripheral A Output register.
0	0	1	Independent Interrupt Input Mode — Set IFR0 on a negative transition of the CA2 input signal. Reading or writing ORA does not clear the CA2 Interrupt flag.
0	1	0	Input Mode — Set CA2 Interrupt flag on a positive transition of the CA2 input signal. Clear IFR0 with a read or write of the Peripheral A Output register.
0	1	1	Independent Interrupt Input Mode — Set IFR0 on a positive transition of the CA2 input signal. Reading or writing ORA does not clear the CA2 Interrupt flag.
1	0	0	Handshake Output Mode — Set CA2 output low on a read or write of the Peripheral A Output register. Reset CA2 high with an active transition on CA1.
1	0	1	Pulse Output Mode — CA2 goes low for one cycle following a read or write of the Peripheral A Output register.
1	1	0	Manual Output Mode — The CA2 output is held low in this mode.
1	1	1	Manual Output Mode — The CA2 output is held high in this mode.

The auxiliary control register is less important than the peripheral control register. Its bits have the following functions (see Figure 1-8):

- Bits 0 and 1, if set, cause the VIA to latch the input data on port A (bit 0) or port B (bit 1) when an active transition occurs on control line 1. This option allows for the case in which the input peripheral provides valid data only briefly, and the data must be saved until the processor has time to handle it.

- Bits 2, 3, and 4 control the operations of the seldom-used shift register. This register provides a simple serial I/O capability, but most designers prefer either to use the serial I/O chips such as the 6551 or 6850 or to provide the entire serial interface in software.

- Bit 5 determines whether timer 2 generates a single time interval (the so-called one-shot mode) or counts pulses on line PB6 (pulse-counting mode).

- Bit 6 determines whether timer 1 generates one time interval (0) or operates continuously (1), reloading its counters from the latches after each interval elapses.

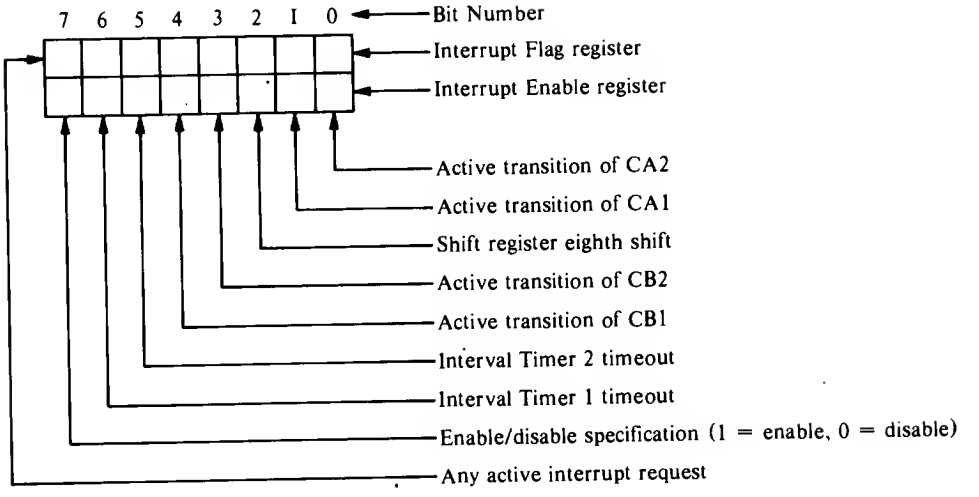
Table 1-14: Operating Modes for Control Line CB2 of a 6522 VIA

PCR7	PCR6	PCR5	Mode
0	0	0	Interrupt Input Mode — Set CB2 Interrupt flag (IFR3) on a negative transition of the CB2 input signal. Clear IFR3 on a read or write of the Peripheral B Output register.
0	0	I	Independent Interrupt Input Mode — Set IFR3 on a negative transition of the CB2 input signal. Reading or writing ORB does not clear the Interrupt flag.
0	I	0	Input Mode — Set CB2 Interrupt flag on a positive transition of the CB2 input signal. Clear the CB2 Interrupt flag on a read or write of ORB.
0	I	I	Independent Input Mode — Set IFR3 on a positive transition of the CB2 input signal. Reading or writing ORB does not clear the CB2 Interrupt flag.
I	0	0	Handshake Output Mode — Set CB2 low on a write ORB operation. Reset CB2 high on an active transition of the CB1 input signal.
I	0	I	Pulse Output Mode — Set CB2 low for one cycle following a write ORB operation.
I	1	0	Manual Output Mode — The CB2 output is held low in this mode.
I	I	I	Manual Output Mode — The CB2 output is held high in this mode.

· Bit 7 determines whether timer 1 generates output pulses on PB7 (a logic 1 generates pulses).

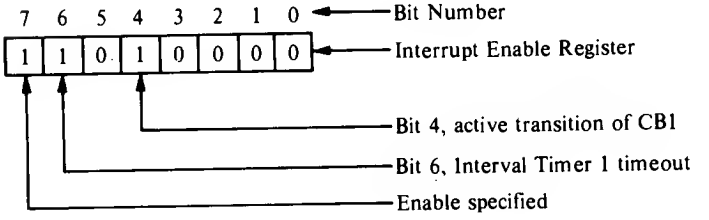
The uses of most of these functions are straightforward. They are not as common as the handshaking functions governed by the peripheral control register.

You can also operate a 6522 VIA in an interrupt-driven mode. Interrupts are enabled or disabled by setting bits in the interrupt enable register (see Figures 1-10 and 1-11) with bit 7 (the enable/disable flag) set (for enabling) or cleared (for disabling). Interrupts can be recognized by examining the interrupt flag register (see Figure 1-9). Table 1-15 summarizes the setting and clearing (resetting) of interrupt flags on the 6522 VIA.



The Interrupt Flag register identifies those interrupts which are active. A 1 in any bit position indicates an active interrupt, whereas a 0 indicates an inactive interrupt.

Figure 1-10: The 6522 VIA's Interrupt Flag and Interrupt Enable Registers



You can selectively enable or disable individual interrupts via the Interrupt Enable register. You enable individual interrupts by writing to the Interrupt Enable register with a 1 in bit 7. Thus you could enable "time out for Timer 1" and "active transitions of signal CB1" by storing $D0_{16}$ in the Interrupt Enable register:

Figure 1-11: A Typical Enabling Operation on the 6522 VIA's Interrupt Enable Register

Table 1-15: A Summary of Conditions for Setting and Resetting Interrupt Flags in the 6522 VIA

	Set by	Cleared by
6	Timeout of Timer 1	Reading Timer 1 Low-Order Counter or writing T1 High-Order Latch
5	Timeout of Timer 2	Reading Timer 2 Low-Order Counter or writing T2 High-Order Counter
4	Active transition of the signal on CB1	Reading from or writing to I/O Port B
3	Active transition of the signal on CB2 (input mode)	Reading from or writing to I/O Port B in input mode only
2	Completion of eight shifts	Reading or writing the Shift register
1	Active transition of the signal on CA1	Reading from or writing to I/O Port A using address 0001_2
0	Active transition of the signal on CA2 (input mode)	Reading from or writing to I/O Port A Output register (ORA) using address 0001_2 in input mode only

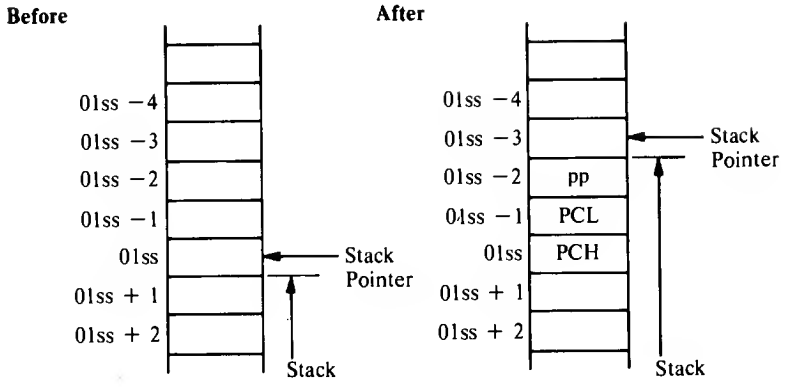
WRITING INTERRUPT- DRIVEN CODE

The 6502 microprocessor responds to an interrupt (either a nonmaskable interrupt, a maskable interrupt that is enabled, or a BRK instruction) as follows:

- By saving the program counter (more significant byte first) and the status register in the stack in the order shown in Figure 1-12. Note that the status register ends up on top of the program counter; the sequence PHP, JSR would produce the opposite order. The program counter value here is the address of the next instruction; there is no offset of 1 as there is with JSR.
- By disabling the maskable interrupt by setting the I flag in the status register.
- By fetching a destination address from a specified pair of memory addresses (see Table 1-16) and placing that destination in the program counter.

Thus, the programmer should consider the following guidelines when writing interrupt-driven code for the 6502:

- The accumulator and index registers must be saved and restored explicitly if the service routine changes them. Only the status register is saved automatically.



ss = Original contents of Stack Pointer
pp = Original contents of Status (P) register
PCH = Original contents of 8 higher order bits of Program Counter
PCL = Original contents of 8 lower order bits of Program Counter

Figure 1-12: The 6502 Microprocessor's Response to an Interrupt

The service routine must save the accumulator before it saves the index registers, since it can only transfer an index register to the stack via the accumulator. Typical saving and restoring sequences are

```
PHA                 ;SAVE ACCUMULATOR IN STACK
TXA                 ;SAVE INDEX REGISTER X
PHA
TYA                 ;SAVE INDEX REGISTER Y
PHA

PLA                 ;RESTORE INDEX REGISTER Y
TAY
PLA                 ;RESTORE INDEX REGISTER X
TAX
PLA                 ;RESTORE ACCUMULATOR FROM STACK
```

The order of the index registers does not matter, as long as the saving and restoring orders are opposites.

· The interrupt need not be reenabled explicitly, since the RTI (Return from Interrupt) instruction restores the old status register as part of its execution. This restores the original state of the Interrupt Disable flag. If you wish to return with interrupts disabled, you can set the Interrupt Disable flag in the stack with the sequence

```
PLA                 ;GET STATUS REGISTER
ORA    #%00000100   ;DISABLE INTERRUPT IN STACK
PHA                 ;PUT STATUS REGISTER BACK IN STACK
```

Table 1-16: Interrupt Vectors for the 6502 Microprocessor.

Source	Address Used (Hexadecimal)
Interrupt Request ($\overline{IRQ}$) and BRK Instruction	FFFE and FFFF
Reset ($\overline{RESET}$)	FFFC and FFFD
Nonmaskable Interrupt ($\overline{NMI}$)	FFFA and FFFB
The addresses are stored in the usual 6502 fashion with the less significant byte at the lower address.	

Note the convenience here of having the status register at the top, rather than underneath the return address.

• If you have code that the processor must execute with interrupts disabled, you can use SEI (Set Interrupt Disable) to disable maskable interrupts and CLI (Clear Interrupt Disable) to enable them afterward. If the section of code could be entered with interrupts either disabled or enabled, you must be sure to restore the original state of the Interrupt Disable flag. That is, you must save and restore the status register as follows:

```
PHP                 ;SAVE OLD INTERRUPT DISABLE
SEI                 ;DISABLE INTERRUPTS
.
.  CODE THAT MUST BE EXECUTED WITH INTERRUPTS DISABLED
.
PLP                 ;RESTORE OLD INTERRUPT DISABLE
```

The alternative (automatically reenabling the interrupts at the end) would cause a problem if the section were entered with the interrupts already disabled.

• If you want to allow the user to select the actual starting address of the service routine, place an indirect jump at the vectored address. That is, the routine starting at the vectored address is simply

```
JMP    (USRINT)     ;GO TO USER-SPECIFIED ADDRESS
```

This procedure increases the interrupt response time by the execution time of an indirect jump (five clock cycles).

• You must remember to save and restore incidental information that is essential for the proper execution of the interrupted program. Such incidental information may include memory locations on page 0, priority registers (particularly if they are write-only), and other status.

• To achieve general reentrancy, you must use the stack for all temporary storage beyond that provided by the registers. As we noted in the discussion of

parameter passing, you can assign space on the stack (NPARAM bytes) with the sequence

```
TSX                 ;MOVE S OVER TO A
TXA
SEC                 ;ASSIGN NPARAM EMPTY BYTES
SBC     #NPARAM     ;A GENERAL WAY TO ADJUST SP
TAX
TXS
```

Later, you can remove the temporary storage area with the sequence

```
TSX                 ;MOVE S OVER TO A
TXA
CLC
ADC     #NPARAM     ;REMOVE NPARAM EMPTY BYTES
TAX
TXS
```

If NPARAM is only 1 or 2, you can replace these sequences with the appropriate number of push and pull instructions in which the data is ignored.

- The service routine should initialize the Decimal Mode flag with either CLD or SED if it uses ADC or SBC instructions. The old value of that flag is saved and restored automatically as part of the status register, but the service routine should not assume a particular value on entry.

MAKING PROGRAMS RUN FASTER

In general, you can make a program run substantially faster by first determining where it is spending its time. This requires that you determine which loops (other than delay routines) the processor is executing most often. Reducing the execution time of a frequently executed loop will have a major effect because of the multiplying factor. It is thus critical to determine how often instructions are being executed and to work on loops in the order of their frequency of execution.

Once you have determined which loops the processor executes most frequently, you can reduce their execution time with the following techniques:

- Eliminate redundant operations. These may include a constant that is being added during each iteration or a special case that is being tested for repeatedly. It may also include a constant value or a memory address that is being fetched each time rather than being stored in a register or used indirectly.
- Use page 0 for temporary data storage whenever possible.
- Reorganize the loop to reduce the number of jump instructions. You can often eliminate branches by changing the initial conditions, reversing the order of

operations, or combining operations. In particular, you may find it helpful to start everything back one step, thus making the first iteration the same as all the others. Reversing the order of operations can be helpful if numerical comparisons are involved, since the equality case may not have to be handled separately. Reorganization may also allow you to combine condition checking inside the loop with the overall loop control.

- Work backward through arrays rather than forward. This allows you to count the index register down to 0 and use the setting of the Zero flag as an exit condition. No explicit comparison is then necessary. Note that you will have to subtract 1 from the base addresses, since 1 is the smallest index that is actually used.

- Increment 16-bit counters and indirect addresses rather than decrementing them. 16-bit numbers are easy to increment, since you can tell if a carry has occurred by checking the less significant byte for 0 afterward. In the case of a decrement, you must check for 0 first.

- Use in-line code rather than subroutines. This will save at least a JSR instruction and an RTS instruction.

- Watch the special uses of the index registers to avoid having to move data between them. The only register that can be used in indirect indexed addressing is register Y; the only register that can be used in indexed indirect addressing or in loading and storing the stack pointer is register X.

- Use the instructions ASL, DEC, INC, LSR, ROL, and ROR to operate directly on data in memory without moving it to a register.

- Use the BIT instruction to test bits 6 or 7 of a memory location without loading the accumulator.

- Use the CPX and CPY instructions to perform comparisons without using the accumulator.

A general way to reduce execution time is to replace long sequences of instructions with tables. A single table lookup can perform the same operation as a sequence of instructions if there are no special exits or program logic involved. The cost is extra memory, but that may be justified if the memory is readily available. If enough memory is available, a lookup table may be a reasonable approach even if many of its entries are repetitive — even if many inputs produce the same output. In addition to its speed, table lookup is easy to program, easy to change, and highly flexible.

MAKING PROGRAMS USE LESS MEMORY[8]

You can make a program use significantly less memory only by identifying common sequences of instructions and replacing those sequences with subroutine calls. The result is a single copy of each sequence. The more instructions you can place in subroutines, the more memory you save. The drawbacks of this approach are that JSR and RTS themselves require memory and take time to execute, and that the subroutines are typically not very general and may be difficult to understand or use. Some sequences of instructions may even be implemented as subroutines in a monitor or in other systems programs that are always resident. Then you can replace those sequences with calls to the systems program as long as the return is handled properly.

Some of the methods that reduce execution time also reduce memory usage. In particular, using page 0, reorganizing loops, working backward through arrays, incrementing 16-bit quantities, operating directly on memory, and using special instructions such as CPX, CPY, and BIT reduce both execution time and memory usage. Of course, using in-line code rather than loops and subroutines reduces execution time but increases memory usage.

Lookup tables generally use extra memory but save execution time. Some ways that you can reduce their memory requirements are by eliminating intermediate values and interpolating the results,[9,10] eliminating redundant values with special tests, and reducing the range of input values. Often you will find that a few prior tests or restrictions will greatly reduce the size of the required table.

REFERENCES

1. Weller, W.J., *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, Ill., 1980.

2. Fischer, W.P., "Microprocessor Assembly Language Draft Standard," *IEEE Computer,* December 1979, pp. 96-109. Further discussions of the standard appear on pp. 79-80 of *IEEE Computer,* April 1980, and on pp. 8-9 of IEEE Computer, May 1981. See also Duncan, F.G., "Level-Independent Notation for Microcomputer Programs," *IEEE Micro,* May 1981, pp. 47-56.

3. Osborne, A. *An Introduction to Microcomputers: Volume 1 — Basic Concepts,* 2nd ed., Berkeley: Osborne/McGraw-Hill, 1980.

4. Ibid.

5. Shankar, K.S., "Data Structures, Types, and Abstractions," *IEEE Computer,* April 1980, pp. 67-77.

6. Ibid.

7. Osborne, A. and G. Kane, *4 and 8-Bit Microprocessor Handbook,* Berkeley: Osborne/McGraw-Hill, 1981, pp. 9-55 to 9-61 (6850 ACIA), Chapter 10 (6500 processors and associated chips).

8. Schember, K.A. and J.R. Rumsey, "Minimal Storage Sorting and Searching Techniques for RAM Applications," *Computer,* June 1977, pp. 92-100.

9. Seim, T.A., "Numerical Interpolation for Microprocessor-Based Systems," *Computer Design,* February 1978, pp. 111-16.

10. Abramovich, A. and T.R. Crawford, "An Interpolating Algorithm for Control Applications on Microprocessors," Proceedings of the 1978 Conference on Industrial Applications of Microprocessors, Philadelphia, Penn., pp. 195-201 (proceedings available from IEEE or IEEE Computer Society).

Two hobby magazines run many articles on 6502 assembly language programming; they are *Compute* (P.O. Box 5406, Greensboro, NC 27403) and *Micro* (P.O. Box 6502, Chelmsford, MA 01824).

Chapter 2 Implementing Additional Instructions And Addressing Modes

This chapter shows how to implement instructions and addressing modes that are not included in the 6502's instruction set. Of course, no instruction set can ever include all possible combinations. Designers must make choices based on how many operation codes are available, how easily an additional combination could be implemented, and how often it would be used. A description of additional instructions and addressing modes does not imply that the basic instruction set is incomplete or poorly designed.

We concentrate our attention on additional instructions and addressing modes that are

- Obvious parallels to those included in the instruction set
- Described in the draft Microprocessor Assembly Language Standard (IEEE Task P694)
- Discussed in Volume 1 of *An Introduction to Microcomputers*[1]
- Implemented on other microprocessors, especially ones that are closely related or partly compatible.[2,3]

This chapter should be of particular interest to those who are familiar with the assembly languages of other computers.

INSTRUCTION SET EXTENSIONS

In describing extensions to the instruction set, we follow the organization suggested in the draft standard for IEEE Task P694.[4] We divide instructions into the following groups (listed in the order in which they are discussed): arithmetic, logical, data transfer, branch, skip, subroutine call, subroutine return, and miscellaneous. Within each type of instruction, we discuss operand types in the

following order: byte (8-bit), word (16-bit), decimal, bit, nibble or digit, and multiple. In describing addressing modes, we use the following order: direct, indirect, immediate, indexed, register, autopreincrement, autopostincrement, autopredecrement, autopostdecrement, indirect preindexed (also called preindexed or indexed indirect), and indirect postindexed (also called postindexed or indirect indexed).

ARITHMETIC INSTRUCTIONS

In this group, we consider addition, addition with carry, subtraction, subtraction in reverse, subtraction with carry (borrow), increment, decrement, multiplication, division, comparison, two's complement (negate), and extension. Instructions that do not obviously fall into a particular category are repeated for convenience.

Addition Instructions (Without Carry)

1. Add memory location ADDR to accumulator.

```
CLC                 ;CLEAR CARRY
ADC     ADDR        ; (A) = (A) + (ADDR)
```

The same approach works for all addressing modes.

2. Add VALUE to accumulator.

```
CLC                 ;CLEAR CARRY
ADC     #VALUE      ; (A) = (A) + VALUE
```

3. Add Carry to accumulator.

```
ADC     #0          ; (A) = (A) + 0 + CARRY
```

4. Decimal add memory location ADDR to accumulator.

```
SED                 ;ENTER DECIMAL MODE
CLC                 ;CLEAR CARRY
ADC     ADDR        ; (A) = (A) + (ADDR) IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

A more general approach restores the original value of the D flag; that is,

```
PHP                 ;SAVE OLD D FLAG
SED                 ;ENTER DECIMAL MODE
CLC                 ;CLEAR CARRY
ADC     ADDR        ; (A) = (A) + (ADDR) IN DECIMAL
PLP                 ;RESTORE OLD D FLAG
```

Note that restoring the status register destroys the carry from the addition.

5. Decimal add VALUE to accumulator.

```
SED                 ;ENTER DECIMAL MODE
CLC                 ;CLEAR CARRY
ADC     #VALUE      ;(A) = (A) + VALUE IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

6. Decimal add Carry to accumulator.

```
SED                 ;ENTER DECIMAL MODE
ADC     #0          ;(A) = (A) + CARRY IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

7. Add index register to accumulator (using memory location ZPAGE).

```
STX     ZPAGE       ;SAVE INDEX REGISTER ON PAGE ZERO
CLC                 ;CLEAR CARRY
ADC     ZPAGE       ;(A) = (A) + (X)
```

This approach works for index register Y also.

8. Add the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) to memory locations SUM and SUM+1 (MSB in SUM+1).

```
CLC                 ;CLEAR CARRY
LDA     SUM
ADC     ADDR        ;ADD LSB'S
STA     SUM
LDA     SUM+1       ;ADD MSB'S WITH CARRY
ADC     ADDR+1
STA     SUM+1
```

9. Add 16-bit number VAL16 (VAL16M more significant byte, VAL16L less significant byte) to memory locations SUM and SUM+1 (MSB in SUM+1).

```
CLC                 ;CLEAR CARRY
LDA     SUM         ;ADD LSB'S WITHOUT CARRY
ADC     #VAL16L
STA     SUM
LDA     SUM+1       ;ADD MSB'S WITH CARRY
ADC     #VAL16
STA     SUM+1
```

Addition Instructions (With Carry)

1. Add Carry to accumulator

```
ADC     #0          ;(A) = (A) + CARRY
```

2. Decimal add VALUE to accumulator with Carry.

```
SED                 ;ENTER DECIMAL MODE
ADC     #VALUE      ;(A) = (A) + VALUE + CARRY IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

3. Decimal add memory location ADDR to accumulator with Carry.

```
SED                 ;ENTER DECIMAL MODE
ADC     ADDR        ;(A) = (A) + (ADDR) + CARRY IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

4. Add the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) to memory locations SUM and SUM+1 (MSB in SUM+1) with Carry.

```
LDA     SUM         ;ADD LSB'S WITH CARRY
ADC     ADDR
STA     SUM
LDA     SUM+1       ;ADD MSB'S WITH CARRY
ADC     ADDR+1
STA     SUM+1
```

5. Add 16-bit number VAL16 (VAL16M more significant byte, VAL16L less significant byte) to memory locations SUM and SUM+1 (MSB in SUM+1) with Carry.

```
LDA     SUM         ;ADD LSB'S WITH CARRY
ADC     VAL16L
STA     SUM
LDA     SUM+1       ;ADD MSB'S WITH CARRY
ADC     ADDR+1
STA     SUM+1
```

Subtraction Instructions (Without Borrow)

1. Subtract memory location ADDR from accumulator.

```
SEC                 ;SET INVERTED BORROW
SBC     ADDR        ;(A) = (A) - (ADDR)
```

The Carry flag acts as an inverted borrow, so it must be set to 1 if its value is to have no effect on the subtraction.

2. Subtract VALUE from accumulator.

```
SEC                 ;SET INVERTED BORROW
SBC     #VALUE      ;(A) = (A) - VALUE
```

3. Subtract inverse of borrow from accumulator.

```
SBC     #0          ;(A) = (A) - (1-CARRY)
```

The result is (A)−1 if Carry is 0 and (A) if Carry is 1.

4. Decimal subtract memory location ADDR from accumulator.

```
SED                 ;ENTER DECIMAL MODE
SEC                 ;SET INVERTED BORROW
SBC     ADDR        ;(A) = (A) - (ADDR) IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

The Carry flag has the same meaning in the decimal mode as in the binary mode.

5. Decimal subtract VALUE from accumulator.

```
SED                 ;ENTER DECIMAL MODE
SEC                 ;SET INVERTED BORROW
SBC     #VALUE      ;(A) = (A) - VALUE IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

6. Subtract the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) from memory locations DIFF and DIFF+1 (MSB in DIFF+1).

```
LDA     DIFF        ;SUBTRACT LSB'S WITH NO BORROW
SEC
SBC     ADDR
STA     DIFF
LDA     DIFF+1      ;SUBTRACT MSB'S WITH BORROW
SBC     ADDR+1
STA     DIFF+1
```

7. Subtract 16-bit number VAL16 (VAL16M more significant byte, VAL16L less significant byte) from memory locations DIFF and DIFF+1 (MSB in DIFF+1).

```
LDA     DIFF        ;SUBTRACT LSB'S WITH NO BORROW
SEC
SBC     #VAL16L
STA     DIFF
LDA     DIFF+1      ;SUBTRACT MSB'S WITH BORROW
SBC     #VAL16M
STA     DIFF+1
```

8. Decimal subtract inverse of borrow from accumulator.

```
SED                 ;ENTER DECIMAL MODE
SBC     #0          ;(A) = (A) - (1-CARRY) IN DECIMAL
CLD                 ;LEAVE DECIMAL MODE
```

Subtraction in Reverse Instructions

1. Subtract accumulator from VALUE and place difference in accumulator.

```
EOR     #$FF        ;ONE'S COMPLEMENT A
CLC
ADC     #1          ;TWO'S COMPLEMENT A
CLC
ADC     #VALUE      ;FORM -A + VALUE
```

or

```
STA     TEMP        ;SAVE A TEMPORARILY
LDA     #VALUE      ;FORM VALUE - A
SEC
SBC     TEMP
```

The Carry acts as an inverted borrow in either method; that is, the Carry is set to 1 if no borrow is necessary.

2. Subtract accumulator from the contents of memory location ADDR and place difference in accumulator.

```
EOR     #$FF        ;ONE'S COMPLEMENT A
CLC
ADC     #1          ;TWO'S COMPLEMENT A
CLC
ADC     ADDR        ;FORM -A + (ADDR)

STA     TEMP        ;SAVE A TEMPORARILY
LDA     ADDR        ;FORM (ADDR) - A
SEC
SBC     TEMP
```

3. Decimal subtract accumulator from VALUE and place difference in accumulator.

```
SED                 ;ENTER DECIMAL MODE
STA     TEMP        ;FORM VALUE - A
LDA     #VALUE
SEC
SBC     TEMP
CLD                 ;LEAVE DECIMAL MODE
```

4. Decimal subtract accumulator from the contents of memory location ADDR and place difference in accumulator.

```
SED                 ;ENTER DECIMAL MODE
STA     TEMP        ;FORM (ADDR) - A
LDA     ADDR
SEC
SBC     TEMP
CLD                 ;LEAVE DECIMAL MODE
```

Subtraction with Borrow (Carry) Instructions

1. Subtract inverse of borrow from accumulator.

```
SBC     #0              ;(A) = (A) - (1-CARRY)
```

2. Decimal subtract VALUE from accumulator with borrow.

```
SED                     ;ENTER DECIMAL MODE
SBC     #VALUE          ;(A) = (A) - VALUE - BORROW IN DECIMAL
CLD                     ;LEAVE DECIMAL MODE
```

3. Decimal subtract memory location ADDR from accumulator with borrow.

```
SED                     ;ENTER DECIMAL MODE
SBC     ADDR            ;(A) = (A) - VALUE - BORROW IN DECIMAL
CLD                     ;LEAVE DECIMAL MODE
```

4. Subtract the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) from memory locations DIFF and DIFF+1 (MSB in DIFF+1) with borrow.

```
LDA     DIFF            ;SUBTRACT LSB'S WITH BORROW
SBC     ADDR
STA     DIFF
LDA     DIFF+1          ;SUBTRACT MSB'S WITH BORROW
SBC     ADDR+1
STA     DIFF+1
```

5. Subtract 16-bit number VAL16 (VAL16M more significant byte, VAL16L less significant byte) from memory locations DIFF and DIFF+1 (MSB in DIFF+1) with borrow.

```
LDA     DIFF            ;SUBTRACT LSB'S WITH BORROW
SBC     VAL16L
STA     DIFF
LDA     DIFF+1          ;SUBTRACT MSB'S WITH BORROW
SBC     VAL16M
STA     DIFF+1
```

Increment Instructions

1. Increment accumulator, setting the Carry flag if the result is zero.

```
CLC                     ;CLEAR CARRY
ADC     #1              ;INCREMENT BY ADDING 1
```

or

```
SEC                     ;SET CARRY
ADC     #0              ;INCREMENT BY ADDING 1
```

2. Increment accumulator without affecting the Carry flag.

```
        TAX             ;MOVE A TO X
        INX             ;INCREMENT X
        TXA
```

INX does not affect the Carry flag; it does, however, affect the Zero flag.

3. Increment stack pointer.

```
        TSX             ;MOVE S TO X
        INX             ;THEN INCREMENT X AND RETURN VALUE
        TXS
```

or

```
        TAX             ;SAVE A
        PLA             ;INCREMENT STACK POINTER
        TXA             ;RESTORE A
```

Remember that PLA affects the Zero and Negative flags.

4. Decimal increment accumulator (add 1 to A in decimal).

```
        SED             ;ENTER DECIMAL MODE
        CLC
        ADC   #1        ;(A) = (A) + 1 DECIMAL
        CLD             ;LEAVE DECIMAL MODE
```

Remember that INC and DEC produce binary results even when the D flag is set.

5. Increment contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        INC   ADDR      ;INCREMENT LSB
        BNE   DONE
        INC   ADDR+1    ;CARRY TO MSB IF LSB GOES TO ZERO
DONE    NOP
```

or

```
        LDA   ADDR      ;INCREMENT LSB
        CLC
        ADC   #1
        STA   ADDR
        LDA   ADDR+1    ;WITH CARRY TO MSB
        ADC   #0
        STA   ADDR+1
```

The first alternative is clearly much shorter.

6. Decimal increment contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        SED             ;ENTER DECIMAL MODE
        LDA   ADDR      ;ADD 1 TO LSB
        CLC
        ADC   #1
```

```
        STA     ADDR
        BCC     DONE
        LDA     ADDR+1          ;CARRY TO MSB IF NECESSARY
        ADC     #0
        STA     ADDR+1
DONE    CLD                     ;LEAVE DECIMAL MODE
```

INC produces a binary result even when the Decimal Mode flag is set. Note that we could eliminate the BCC instruction from the program without affecting the result, but the change would increase the average execution time.

Decrement Instructions

1. Decrement accumulator, clearing the Carry flag if the result is FF_{16}.

```
        SEC                     ;SET INVERTED BORROW
        SBC     #1              ;DECREMENT BY SUBTRACTING 1
```

or

```
        CLC                     ;CLEAR INVERTED BORROW
        SBC     #0              ;DECREMENT BY SUBTRACTING 1
```

or

```
        CLC                     ;CLEAR CARRY
        ADC     #$FF            ;DECREMENT BY ADDING -1
```

2. Decrement accumulator without affecting the Carry flag.

```
        TAX                     ;MOVE A TO X
        DEX                     ;DECREMENT X
        TXA
```

DEX does not affect the Carry flag; it does, however, affect the Zero flag.

3. Decrement stack pointer.

```
        TSX                     ;MOVE S TO X
        DEX                     ;THEN DECREMENT X AND RETURN VALUE
        TXS
```

You can also decrement the stack pointer with PHA or PHP, neither of which affects any flags.

4. Decimal decrement accumulator (subtract 1 from A in decimal).

```
        SED                     ;ENTER DECIMAL MODE
        SEC
        SBC     #1              ;(A) = (A) - 1 DECIMAL
        CLD                     ;LEAVE DECIMAL MODE
```

5. Decrement contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        LDA     ADDR        ;IS LSB ZERO?
        BNE     DECLSB
        DEC     ADDR+1      ;YES, BORROW FROM MSB
DECLSB  DEC     ADDR        ;BEFORE DECREMENTING LSB
```

Decrementing a 16-bit number is significantly more difficult than incrementing one. In fact, incrementing is not only faster but also leaves the accumulator unchanged; of course, one could replace LDA with LDX, LDY, or the sequence INC, DEC. An alternative that uses no registers is

```
        INC     ADDR        ;IS LSB ZERO?
        DEC     ADDR
        BNE     DECLSB
        DEC     ADDR+1      ;YES, BORROW FROM MSB
DECLSB  DEC     ADDR        ;BEFORE DECREMENTING LSB
```

6. Decimal decrement contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        SED                 ;ENTER DECIMAL MODE
        LDA     ADDR        ;SUBTRACT 1 FROM LSB
        SEC
        SBC     #1
        STA     ADDR
        BCS     DONE
        LDA     ADDR+1      ;BORROW FROM MSB IF NECESSARY
        SBC     #0
        STA     ADDR+1
DONE    CLD                 ;LEAVE DECIMAL MODE
```

DEC produces a binary result even when the Decimal Mode flag is set. Note that we could eliminate the BCS instruction from the program without affecting the result, but the change would increase the average execution time.

Multiplication Instructions

1. Multiply accumulator by 2.

```
        ASL     A           ;MULTIPLY BY SHIFTING LEFT
```

The following version places the Carry (if any) in Y.

```
        LDY     #0          ;ASSUME MSB = 0
        ASL     A           ;MULTIPLY BY SHIFTING LEFT
        BCC     DONE
        INY                 ;AND MOVING CARRY TO Y
DONE    NOP
```

2. Multiply accumulator by 3 (using ADDR for temporary storage).

```
STA     ADDR            ;SAVE A
ASL     A               ;2 X A
ADC     ADDR            ;3 X A
```

3. Multiply accumulator by 4.

```
ASL     A               ;2 X A
ASL     A               ;4 X A
```

We can easily extend cases 1, 2, and 3 to multiplication by other small integers.

4. Multiply an index register by 2.

```
TAX                     ;MOVE TO A
ASL     A               ;MULTIPLY BY SHIFTING LEFT
TXA                     ;RETURN RESULT
```

5. Multiply the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 2.

```
ASL     ADDR            ;MULTIPLY BY SHIFTING LEFT
ROL     ADDR+1          ;AND MOVING CARRY OVER TO MSB
```

6. Multiply the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 4.

```
ASL     ADDR            ;MULTIPLY BY SHIFTING LEFT
ROL     ADDR+1          ;AND MOVING CARRY OVER TO MSB
ASL     ADDR            ;THEN MULTIPLY AGAIN
ROL     ADDR+1
```

Eventually, of course, moving one byte to the accumulator, shifting the accumulator, and storing the result back in memory becomes faster than leaving both bytes in memory.

Division Instructions

1. Divide accumulator by 2 unsigned.

```
LSR     A               ;DIVIDE BY SHIFTING RIGHT
```

2. Divide accumulator by 4 unsigned.

```
LSR     A               ;DIVIDE BY SHIFTING RIGHT
LSR     A
```

3. Divide accumulator by 2 signed.

```
TAX                     ;SAVE ACCUMULATOR
ASL     A               ;MOVE SIGN TO CARRY
TXA                     ;RESTORE ACCUMULATOR
ROR     A               ;SHIFT RIGHT BUT PRESERVE SIGN
```

The second instruction moves the original sign bit (bit 7) to the Carry flag, so the final rotate can preserve it. This is known as an *arithmetic shift,* since it preserves the sign of the number while reducing its magnitude. The fact that the sign bit is copied to the right is known as *sign extension.*

4. Divide the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 2 unsigned.

```
        LSR     ADDR+1          ;DIVIDE BY SHIFTING RIGHT
        ROR     ADDR            ;AND MOVING CARRY OVER TO LSB
```

5. Divide the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 2 signed.

```
        LDA     ADDR+1          ;MOVE SIGN TO CARRY
        ASL     A
        ROR     ADDR+1          ;DIVIDE BY SHIFTING RIGHT WITH SIGN
        ROR     ADDR            ;AND MOVING CARRY OVER TO LSB
```

Comparison Instructions

1. Compare VALUE with accumulator bit by bit, setting each bit position that is different.

```
        EOR     #VALUE
```

Remember, the EXCLUSIVE OR of two bits is 1, if and only if the two bits are different.

2. Compare memory locations ADR1 and ADR1+1 (MSB in ADR1+1) with memory locations ADR2 and ADR2+1 (MSB in ADR2+1). Set Carry if the first operand is greater than or equal to the second one (that is, if ADR1 and ADR1+1 contain a 16-bit unsigned number greater than or equal to the contents of ADR2 and ADR2+1). Clear Carry otherwise. Set the Zero flag if the two operands are equal and clear it otherwise.

```
        LDA     ADR1+1          ;COMPARE MSB'S
        CMP     ADR2+1
        BCC     DONE            ;CLEAR CARRY, ZERO IF 2ND IS LARGER
        BNE     DONE            ;SET CARRY, CLEAR ZERO IF 1ST LARGER
        LDA     ADR1            ;IF MSB'S EQUAL, COMPARE LSB'S
        CMP     ADR2            ;CLEAR CARRY IF 2ND IS LARGER
DONE    NOP
```

3. Compare memory locations ADR1 and ADR1+1 (MSB in ADR1+1) with the 16-bit number VAL16 (VAL16M more significant byte, VAL16L less significant byte). Set Carry if the contents of ADR1 and ADR1+1 are greater than or

equal to VAL16 in the unsigned sense. Clear Carry otherwise. Set the Zero flag if the contents of ADR1 and ADR1+1 are equal to VAL16, and clear it otherwise.

```
        LDA     ADR1+1          ;COMPARE MSB'S
        CMP     #VAL16M
        BCC     DONE            ;CLEAR CARRY, ZERO IF VAL16 LARGER
        BNE     DONE            ;SET CARRY, CLEAR ZERO IF DATA LARGER
        LDA     ADR1            ;IF MSB'S EQUAL, COMPARE LSB'S
        CMP     #VAL16L         ;CLEAR CARRY IF VAL16 LARGER
DONE    NOP
```

4. Compare memory locations ADR1 and ADR1+1 (MSB in ADR1+1) with memory locations ADR2 and ADR2+1 (MSB in ADR2+1). Set Carry if the first operand is greater than or equal to the second one in the unsigned sense.

```
        LDA     ADR1            ;COMPARE LSB'S
        CMP     ADR2
        LDA     ADR1+1          ;SUBTRACT MSB'S WITH BORROW
        SBC     ADR2+1
```

We use SBC on the more significant bytes in order to include the borrow from the less significant bytes. This sequence destroys the value in A and sets the Zero flag only from the final subtraction.

5. Compare memory locations ADR1 and ADR1+1 (MSB in ADR1+1) with the 16-bit number VAL16 (VAL16M more significant byte, VAL16L less significant byte). Set Carry if the contents of ADR1 and ADR1+1 are greater than or equal to VAL16 in the unsigned sense.

```
        LDA     ADR1            ;COMPARE LSB'S
        CMP     VAL16L
        LDA     ADR1+1          ;SUBTRACT MSB'S WITH BORROW
        SBC     VAL16M
```

If you want to set the Carry if the contents of ADR1 and ADR1+1 are greater than VAL16, perform the comparison with VAL16+1.

6. Compare stack pointer with the contents of memory location ADDR. Set Carry if the stack pointer is greater than or equal to the contents of the memory location in the unsigned sense. Clear Carry otherwise. Set the Zero flag if the two values are equal and clear it otherwise.

```
        TSX                     ;MOVE STACK POINTER TO X
        CPX     ADDR            ;AND THEN COMPARE
```

7. Compare stack pointer with the 8-bit number VALUE. Set Carry if the stack pointer is greater than or equal to VALUE in the unsigned sense. Clear Carry otherwise. Set the Zero flag if the two values are equal and clear it otherwise.

```
        TSX                     ;MOVE STACK POINTER TO X
        CPX     #VALUE          ;AND THEN COMPARE
```

8. Block comparison. Compare accumulator with memory bytes starting at address BASE and continuing until either a match is found (indicated by Carry=1) or until a byte counter in memory location COUNT reaches zero (indicated by Carry=0).

```
          LDY    COUNT        ;GET COUNT
          BEQ    NOTFND       ;EXIT IF COUNT IS ZERO
          LDX    #0           ;START INDEX AT ZERO
CMPBYT    CMP    BASE,X       ;CHECK CURRENT BYTE
          BEQ    DONE         ;DONE IF MATCH FOUND (CARRY = 1)
          INX                 ;OTHERWISE, PROCEED TO NEXT BYTE
          DEY
          BNE    CMPBYT       ;IF ANY ARE LEFT
NOTFND    CLC                 ;OTHERWISE, EXIT CLEARING CARRY
DONE      NOP
```

Remember, comparing two equal numbers sets the Carry flag.

Two's Complement (Negate) Instructions

1. Negate accumulator.

```
          EOR    #$FF         ;ONE'S COMPLEMENT
          CLC
          ADC    #1           ;TWO'S COMPLEMENT
```

The two's complement is the one's complement plus 1.

```
          STA    TEMP         ;ALTERNATIVE IS 0 - (A)
          LDA    #0
          SEC
          SBC    TEMP
```

2. Negate memory location ADDR.

```
          LDA    #0           ;FORM 0 - (ADDR)
          SEC
          SBC    ADDR
          STA    ADDR
```

3. Negate memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
          LDA    ADDR         ;ONE'S COMPLEMENT LSB
          EOR    #$FF
          CLC                 ;ADD 1 FOR TWO'S COMPLEMENT
          ADC    #1
          STA    ADDR
          LDA    ADDR+1       ;ONE'S COMPLEMENT MSB
          EOR    #$FF
          ADC    #0           ;ADD CARRY FOR TWO'S COMPLEMENT
          STA    ADDR+1
```

or

```
LDA     #0              ;FORM 0 - (ADDR+1) (ADDR)
SEC
SBC     ADDR            ;SUBTRACT LSB'S WITHOUT BORROW
STA     ADDR
LDA     #0              ;SUBTRACT MSB'S WITH BORROW
SBC     ADDR+1
STA     ADDR+1
```

4. Nine's complement accumulator (that is, replace A with 99−A).

```
STA     TEMP            ;FORM 99-A
LDA     #$99
SEC
SBC     TEMP
```

There is no need to bother with the decimal mode, since 99−A is always a valid BCD number if A originally contained a valid BCD number.

5. Ten's complement accumulator (that is, replace A with 100−A).

```
SED                     ;ENTER DECIMAL MODE
STA     TEMP            ;FORM 100-A
LDA     #0
SEC
SBC     TEMP
CLD                     ;LEAVE DECIMAL MODE
```

Extend Instructions

1. Extend accumulator to a 16-bit unsigned number in memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
STA     ADDR            ;8-BIT MOVE
LDA     #0              ;EXTEND TO 16 BITS WITH 0'S
STA     ADDR+1
```

2. Extend accumulator to a 16-bit signed number in memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
STA     ADDR            ;8-BIT MOVE
ASL     A               ;MOVE SIGN BIT TO CARRY
LDA     #$FF            ;(A) = -1 + SIGN BIT
ADC     #0
EOR     #$FF            ;(A) = -SIGN BIT
STA     ADDR+1          ;SET MSB TO -SIGN BIT
```

The result of the calculation is −(−1+SIGN BIT)−1=−SIGN BIT. That is, (ADDR+1) = 00 if A was positive and FF_{16} if A was negative. An alternative is

```
        STA     ADDR            ;8-BIT MOVE
        LDX     #$FF            ;(X) = -1
        ASL     A
        BCS     STRSGN
        INX                     ;(X) = -1+(1 - SIGN BIT) = -SIGN BIT
STRSGN  STX     ADDR+1          ;SET MSB TO -SIGN BIT
```

3. Extend bit 0 of accumulator across entire accumulator; that is, (A) = 00 if bit 0 = 0 and FF_{16} if bit 0 = 1.

```
        LSR     A               ;CARRY = BIT 0
        LDA     #$FF            ;(A) = -1 + BIT 0
        ADC     #0
        EOR     #$FF            ;(A) = -BIT 0
```

As in case 2, the result we want is −1 if the specified bit is 1 and 0 if the specified bit is 0. That is, we want the negative of the original bit value. The sequence LDA #$FF, ADC #0 obviously produces the result −1+Carry. The one's complement then gives us the negative of what we had minus 1 (or 1−Carry−1 = −Carry).

4. Sign function. Replace the value in the accumulator by 00 if it is positive and by FF_{16} if it is negative.

```
        ASL     A               ;MOVE SIGN BIT TO CARRY
        LDA     #$FF            ;(A) = -1 + SIGN BIT
        ADC     #0
        EOR     #$FF            ;(A) = -SIGN BIT
```

5. Sign function of a memory location. Set accumulator to 00 if memory location ADDR is positive and to FF_{16} if it is negative.

```
        LDX     #$FF            ;ASSUME NEGATIVE
        LDA     ADDR            ;IS (ADDR) POSITIVE?
        BMI     DONE
        INX                     ;YES, SET SIGN TO ZERO
DONE    TXA
```

The approach shown in case 4 can also be used.

LOGICAL INSTRUCTIONS

In this group, we consider logical AND, logical OR, logical EXCLUSIVE OR, logical NOT (complement), shift, rotate, and test instructions.

Logical AND Instructions

1. Clear bit of accumulator.

```
AND     #MASK          ;CLEAR BIT BY MASKING
```

MASK has 0 bits in the positions to be cleared and 1 bits in the positions that are to be left unchanged. For example,

```
AND     #%11011011    ;CLEAR BITS 2 AND 5
```

Remember, logically ANDing a bit with 1 leaves it unchanged.

2. Bit test-set the flags according to the value of a bit of memory location ADDR.

Bits 0 through 5

```
LDA     #MASK
BIT     ADDR           ;TEST BIT OF ADDR
```

MASK should have a 1 in the position to be tested and 0s everywhere else. The Zero flag will be set to 1 if the bit tested is 0 and to 0 if the bit tested is 1.

Bits 6 or 7

```
BIT     ADDR           ;TEST BITS 6 AND 7 OF ADDR
```

This single instruction sets the Negative flag to bit 7 of ADDR and the Overflow flag to bit 6 of ADDR, regardless of the value in the accumulator. Note that the flags are not inverted as the Zero flag is in normal masking.

3. Logical AND immediate with condition codes (flags). Logically AND a byte of immediate data with the contents of the status register, clearing those flags that are logically ANDed with 0s. This instruction is implemented on the 6809 microprocessor.

```
PHP                    ;MOVE STATUS TO A
PLA
AND     #MASK          ;CLEAR FLAGS
PHA                    ;RETURN RESULT TO STATUS
PLP
```

Logical OR Instructions

1. Set bit of accumulator.

```
ORA     #MASK          ;SET BIT BY MASKING
```

MASK has 1 bits in the positions to be set and 0 bits in the positions that are to be left unchanged. For example,

```
ORA     #%00010010    ;SET BITS 1 AND 4
```

Remember, logically ORing a bit with 0 leaves it unchanged.

2. Test memory locations ADDR and ADDR+1 for 0. Set the Zero flag if both bytes are 0.

```
LDA     ADDR          ;TEST 16-BIT NUMBER FOR ZERO
ORA     ADDR+1
```

The Zero flag is set if and only if both bytes of the 16-bit number are 0. The other flags are also changed.

3. Logical OR immediate with condition codes (flags). Logically OR a byte of immediate data (MASK) with the contents of the status register, setting those flags that are logically ORed with 1s. This instruction is implemented on the 6809 microprocessor.

```
PHP                   ;MOVE STATUS TO A
PLA
ORA     #MASK         ;SET FLAGS
PHA                   ;RETURN RESULT TO STATUS
PLP
```

Logical EXCLUSIVE OR Instructions

1. Complement bit of accumulator.

```
EOR     #MASK         ;COMPLEMENT BIT BY MASKING
```

MASK has 1 bits in the positions to be complemented and 0 bits in the positions that are to be left unchanged. For example,

```
EOR     #%11000000    ;COMPLEMENT BITS 6 AND 7
```

Remember, logically EXCLUSIVE ORing a bit with 0 leaves it unchanged.

2. Complement accumulator, setting flags.

```
EOR     #%11111111    ;COMPLEMENT ACCUMULATOR
```

Logically EXCLUSIVE ORing the accumulator with all 1s inverts all the bits.

3. Compare memory location ADDR with accumulator bit by bit, setting each bit position that is different.

```
EOR     ADDR          ;BIT-BY-BIT COMPARISON
```

The EXCLUSIVE OR function is the same as a "not equal" function. Note that the Negative (Sign) flag is 1 if the two operands have different values in bit position 7.

4. Add memory location ADDR to accumulator logically (i.e., without any carries between bit positions).

```
EOR     ADDR            ;LOGICAL ADDITION
```

The EXCLUSIVE OR function is also the same as a bit by bit sum with no carries. Logical sums are often used to form checksums and error-detecting or error-correcting codes.

Logical NOT instructions

1. Complement accumulator, setting flags.

```
EOR     #$FF            ;COMPLEMENT ACCUMULATOR
```

Logically EXCLUSIVE ORing with all 1s inverts all the bits.

2. Complement bit of accumulator.

```
EOR     #MASK           ;COMPLEMENT BIT BY MASKING
```

MASK has 1 bits in the positions to be complemented and 0 bits in the positions that are to be left unchanged. For example,

```
EOR     #%01010001  ;COMPLEMENT BITS 0, 4, AND 6
```

Remember, logically EXCLUSIVE ORing a bit with 0 leaves it unchanged.

3. Complement a memory location.

```
LDA     ADDR
EOR     #$FF            ;COMPLEMENT
STA     ADDR
```

4. Complement bit 0 of a memory location.

```
INC     ADDR            ;COMPLEMENT BY INCREMENTING
```

or

```
DEC     ADDR            ;COMPLEMENT BY DECREMENTING
```

Either of these instructions may, of course, affect the other bits in the memory location. The final value of bit 0, however, will surely be 0 if it was originally 1 and 1 if it was originally 0.

5. Complement digit of accumulator.

- Less significant digit

```
EOR     #%00001111  ;COMPLEMENT LESS SIGNIFICANT 4 BITS
```

- More significant digit

```
EOR     #%11110000  ;COMPLEMENT MORE SIGNIFICANT 4 BITS
```

These procedures are useful if the accumulator contains a decimal digit in negative logic (e.g., the input from a typical ten-position rotary or thumbwheel switch).

6. Complement Carry flag.

```
        ROR     A               ;MOVE CARRY TO BIT 7 OF A
        EOR     #$FF            ;COMPLEMENT ALL OF A
        ROL     A               ;MOVE COMPLEMENTED CARRY BACK
```

Other combinations such as ROL, EOR, ROR, or ROR, EOR, ASL will work just as well. We could leave the accumulator intact by saving it in the stack originally and restoring it afterward.

An alternative that does not affect the accumulator is

```
        BCC     SETCAR
        CLC                     ;CLEAR CARRY IF IT WAS SET
        BCC     DONE
SETCAR  SEC                     ;SET CARRY IF IT WAS CLEARED
DONE    NOP
```

Shift Instructions

1. Shift accumulator right arithmetically, preserving the sign bit.

```
        TAX                     ;SAVE ACCUMULATOR
        ASL     A               ;MOVE SIGN BIT TO CARRY
        TXA
        ROR     A               ;SHIFT RIGHT, PRESERVING SIGN
```

We need a copy of the sign bit for an arithmetic shift. Of course, we could use a memory location for temporary storage instead of the index register.

2. Shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) left logically.

```
        ASL     ADDR            ;SHIFT LSB LEFT LOGICALLY
        ROL     ADDR+1          ;AND MOVE CARRY OVER TO MSB
```

The key point here is that we must shift the more significant byte circularly (i.e., rotate it). The first 8-bit shift moves one bit (the least significant bit for a right shift and the most significant bit for a left shift) to the Carry. The 8-bit rotate then moves that bit from the Carry into the other half of the word.

3. Shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) right logically.

```
        LSR     ADDR+1          ;SHIFT MSB RIGHT LOGICALLY
        ROR     ADDR            ;AND MOVE CARRY OVER TO LSB
```

4. Shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) right arithmetically.

```
        LDA     ADDR+1          ;MOVE SIGN BIT TO CARRY
        ASL     A
        ROR     ADDR+1          ;SHIFT MSB RIGHT ARITHMETICALLY
        ROR     ADDR            ;AND MOVE CARRY OVER TO LSB
```

5. Digit shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) left; that is, shift the 16-bit number left 4 bits logically.

```
        LDX     #4              ;NUMBER OF SHIFTS = 4
        LDA     ADDR            ;MOVE LSB TO A
SHFT1   ASL     A               ;SHIFT LSB LEFT LOGICALLY
        ROL     ADDR+1          ;AND MOVE CARRY OVER TO MSB
        DEX
        BNE     SHFT1           ;COUNT BITS
        STA     ADDR            ;RETURN LSB TO ADDR
```

A shorter but slower version that does not use the accumulator is

```
        LDX     #4              ;NUMBER OF SHIFTS = 4
SHFT1   ASL     ADDR            ;SHIFT LSB LEFT LOGICALLY
        ROL     ADDR+1          ;AND MOVE CARRY OVER TO MSB
        DEX
        BNE     SHFT1           ;COUNT SHIFTS
```

6. Digit shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) right; that is, shift the 16-bit number right 4 bits logically.

```
        LDX     #4              ;NUMBER OF SHIFTS = 4
        LDA     ADDR            ;MOVE LSB TO A
SHFT1   LSR     ADDR+1          ;SHIFT MSB RIGHT LOGICALLY
        ROR     A               ;AND MOVE CARRY OVER TO LSB
        DEX
        BNE     SHFT1           ;COUNT SHIFTS
        STA     ADDR            ;RETURN LSB TO ADDR
```

A shorter but slower version that does not use the accumulator is

```
        LDX     #4              ;NUMBER OF SHIFTS = 4
SHFT1   LSR     ADDR+1          ;SHIFT MSB RIGHT LOGICALLY
        ROR     ADDR            ;AND MOVE CARRY OVER TO LSB
        DEX
        BNE     SHFT1           ;COUNT SHIFTS
```

7. Normalize memory locations ADDR and ADDR+1 (MSB in ADDR+1); that is, shift the 16-bit number left until the most significant bit is 1. Do not shift at all if the entire number is 0.

```
        LDA     ADDR+1          ;EXIT IF NUMBER ALREADY NORMALIZED
        BMI     DONE
        ORA     ADDR            ;OR IF ENTIRE NUMBER IS ZERO
        BEQ     DONE
        LDA     ADDR            ;MOVE LSB TO A
```

```
SHIFT   ASL   A          ;SHIFT LSB LEFT LOGICALLY 1 BIT
        ROL   ADDR+1     ;AND MOVE CARRY OVER TO MSB
        BPL   SHIFT      ;CONTINUE UNTIL MSB IS 1
        STA   ADDR       ;RETURN LSB TO ADDR
DONE    NOP
```

Rotate Instructions

A rotate through or with Carry acts as if the data were arranged in a circle with its least significant bit connected to its most significant bit through the Carry flag. A rotate without Carry differs in that it acts as if the least significant bit of the data were connected directly to the most significant bit.

1. Rotate memory locations ADDR and ADDR+1 (MSB in ADDR+1) right 1 bit position through Carry.

```
        ROR   ADDR+1     ;ROTATE BIT 8 TO CARRY
        ROR   ADDR       ;AND ON IN TO BIT 7
```

2. Rotate memory locations ADDR and ADDR+1 (MSB in ADDR+1) right 1 bit position without Carry.

```
        LDA   ADDR       ;CAPTURE BIT 0 IN CARRY
        ROR   A
        ROR   ADDR+1     ;ROTATE MSB WITH BIT 0 ENTERING AT LEFT
        ROR   ADDR       ;ROTATE LSB
```

3. Rotate memory locations ADDR and ADDR+1 (MSB in ADDR+1) left 1 bit position through Carry.

```
        ROL   ADDR       ;ROTATE BIT 7 TO CARRY
        ROL   ADDR+1     ;AND ON IN TO BIT 8
```

4. Rotate memory locations ADDR and ADDR+1 (MSB in ADDR+1) left 1 bit position without Carry.

```
        LDA   ADDR+1     ;CAPTURE BIT 15 IN CARRY
        ROL   A
        ROL   ADDR       ;ROTATE LSB WITH BIT 15 ENTERING AT RIGHT
        ROL   ADDR+1
```

Test Instructions

1. **Test accumulator**. Set flags according to the value in the accumulator without changing that value.

```
        TAX              ;MOVE AND SET FLAGS
```

or

```
        TAY              ;MOVE AND SET FLAGS
```

The following alternative does not affect either index register.

```
CMP     #0              ;TEST ACCUMULATOR
```

The instructions AND #$FF or ORA #0 would also do the job without affecting the Carry (CMP #0 sets the Carry flag).

2. **Test index register**. Set flags according to the value in an index register without changing that value.

```
CPX     #0              ;CHECK VALUE IN INDEX REGISTER
```

3. **Test memory location**. Set flags according to the value in memory location ADDR without changing that value.

```
INC     ADDR            ;CHECK VALUE IN MEMORY LOCATION
DEC     ADDR
```

4. **Test a pair of memory locations**. Set the Zero flag according to the value in memory locations ADDR and ADDR+1.

```
LDA     ADDR            ;TEST 16-BIT NUMBER FOR ZERO
ORA     ADDR+1
```

This sequence sets the Zero flag to 1 if and only if both bytes of the 16-bit number are 0. This procedure can readily be extended to handle numbers of any length.

5. **Test bit of accumulator**.

```
AND     #MASK           ;TEST BIT BY MASKING
```

MASK has a 1 bit in the position to be tested and 0 bits elsewhere. The instruction sets the Zero flag to 1 if the tested bit position contains 0 and to 0 if the tested bit position contains 1. For example,

```
AND     #%00001000      ;TEST BIT 3 BY MASKING
```

The result is 0 if bit 3 of A is 0 and 00001000 (binary) if bit 3 of A is 1. So the Zero flag ends up containing the logical complement of bit 3.

6. **Compare memory location ADDR with accumulator bit by bit.** Set each each bit position that is different.

```
EOR     ADDR            ;BIT-BY-BIT COMPARISON
```

The EXCLUSIVE OR function is the same as a "not equal" function.

DATA TRANSFER INSTRUCTIONS

In this group, we consider load, store, move, exchange, clear, and set instructions.

Load Instructions

1. Load accumulator indirect from address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;AVOID INDEXING
LDA     (PGZRO),Y       ;LOAD INDIRECT INDEXED
```

The only instruction that has true indirect addressing is JMP. However, you can produce ordinary indirect addressing by using the postindexed (indirect indexed) addressing mode with index register Y set to 0.

An alternative approach is to clear index register X and use preindexing.

```
LDX     #0              ;AVOID INDEXING
LDA     (PGZRO,X)       ;LOAD INDEXED INDIRECT
```

The advantage of the first approach is that one can index from the indirect address with Y. For example, we could load addresses POINTL and POINTH indirectly from the address in memory locations PGZRO and PGZRO+1 as follows:

```
LDY     #0              ;AVOID INDEXING
LDA     (PGZRO),Y       ;GET LSB OF ADDRESS INDIRECTLY
STA     POINTL
INY                     ;GET MSB OF ADDRESS INDIRECTLY
LDA     (PGZRO),Y
STA     POINTH
```

2. Load index register X indirect from address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;AVOID INDEXING
LDA     (PGZRO),Y       ;LOAD ACCUMULATOR INDIRECT INDEXED
TAX
```

Only the accumulator can be loaded using the indirect modes, but its contents can be transferred easily to an index register.

3. Load index register Y indirect from address in memory locations PGZRO and PGZRO+1.

```
LDX     #0              ;AVOID INDEXING
LDA     (PGZRO,X)       ;LOAD ACCUMULATOR INDEXED INDIRECT
TAY
```

4. Load stack pointer immediate with the 8-bit number VALUE.

```
LDX     #VALUE          ;INITIALIZE STACK POINTER
TXS
```

Only index register X can be transferred to or from the stack pointer.

5. Load stack pointer direct from memory location ADDR.

```
LDX     ADDR            ;INITIALIZE STACK POINTER
TXS
```

6. Load status register immediate with the 8-bit number VALUE.

```
LDA     #VALUE          ;GET THE VALUE
PHA                     ;TRANSFER IT THROUGH STACK
PLP
```

This procedure allows the user of a computer system to initialize the status register for debugging or testing purposes.

7. Load status register direct from memory location ADDR.

```
LDA     ADDR            ;GET THE INITIAL VALUE
PHA                     ;TRANSFER IT THROUGH STACK
PLP
```

8. Load index register from stack.

```
PLA                     ;TRANSFER STACK TO X THROUGH A
TAX
```

If you are restoring values from the stack, you must restore X and Y before A, since there is no direct path from the stack to X or Y.

9. Load memory locations PGZRO and PGZRO+1 (a pointer on page 0) with ADDR (ADDRH more significant byte, ADDRL less significant byte).

```
LDA     #ADDRL          ;INITIALIZE LSB
STA     PGZRO
LDA     #ADDRH          ;INITIALIZE MSB
STA     PGZRO+1
```

There is no simple way to initialize the indirect addresses that must be saved on page 0.

Store Instructions

1. Store accumulator indirect at address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;AVOID INDEXING
STA     (PGZRO),Y       ;STORE INDIRECT INDEXED
```

or

```
LDX     #0              ;AVOID INDEXING
STA     (PGZRO,X)       ;STORE INDEXED INDIRECT
```

2. Store index register X indirect at address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;AVOID INDEXING
TXA                     ;STORE X INDIRECT INDEXED THROUGH A
STA     (PGZRO),Y
```

3. Store index register Y indirect at address in memory locations PGZRO and PGZRO+1.

```
LDX     #0              ;AVOID INDEXING
TYA                     ;STORE Y INDEXED INDIRECT THROUGH A
STA     (PGZRO,X)
```

4. Store stack pointer in memory location ADDR.

```
TSX                     ;STORE S THROUGH X
STX     ADDR
```

5. Store status register in memory location ADDR.

```
PHP                     ;STORE P THROUGH STACK AND A
PLA
STA     ADDR
```

6. Store index register in stack.

```
TXA                     ;STORE X (OR Y) IN STACK VIA A
PHA
```

If you are saving values in the stack, you must save A before X or Y, since there is no direct path from X or Y to the stack.

Move Instructions

1. Transfer accumulator to status register.

```
PHA                     ;TRANSFER THROUGH STACK
PLP
```

2. Transfer status register to accumulator.

```
PHP                     ;TRANSFER THROUGH STACK
PLA
```

3. Transfer index register X to index register Y.

```
TXA                     ;TRANSFER THROUGH ACCUMULATOR
TAY
```

or without changing the accumulator

```
STX     TEMP            ;TRANSFER THROUGH MEMORY
LDY     TEMP
```

4. Transfer accumulator to stack pointer.

```
TAX                     ;TRANSFER THROUGH X REGISTER
TXS
```

5. Transfer stack pointer to accumulator.

```
TSX                     ;TRANSFER THROUGH X REGISTER
TXA
```

6. Move the contents of memory locations ADDR and ADDR+1 (MSB in ADDR+1) to the program counter.

```
        JMP     (ADDR)          ;JUMP INDIRECT
```

Note that JMP with indirect addressing loads the program counter with the contents of memory locations ADDR and ADDR+1; it acts more like LDA with direct addressing than like LDA with indirect (indexed) addressing.

7. Block move. Transfer data from addresses starting at the one in memory locations SORCE and SORCE+1 (on page 0) to addresses starting at the one in memory locations DEST and DEST+1 (on page 0). Register Y contains the number of bytes to be transferred.

```
MOVBYT  DEY                     ;TEST NUMBER OF BYTES
        LDA     (SORCE),Y       ;GET A BYTE FROM SOURCE
        STA     (DEST),Y        ;MOVE TO DESTINATION
        TYA
        BNE     MOVBYT
```

We assume here that the addresses do not overlap and that the initial value of Y is 1 or greater. Chapter 5 contains a more general block move.

The program becomes simpler if we reduce the base addresses by 1. That is, let memory locations SORCE and SORCE+1 contain an address one less than the lowest address in the source area, and let memory locations DEST and DEST+1 contain an address one less than the lowest address in the destination area. Now we can exit when Y is decremented to 0.

```
MOVBYT  LDA     (SORCE),Y       ;GET A BYTE FROM SOURCE
        STA     (DEST),Y        ;MOVE BYTE TO DESTINATION
        DEY
        BNE     MOVBYT          ;COUNT BYTES
```

The 0 index value is never used.

8. Move multiple (fill). Place the contents of the accumulator in memory locations starting at the one in memory locations PGZRO and PGZRO+1.

```
FILBYT  DEY
        STA     (PGZRO),Y       ;FILL A BYTE
        INY
        DEY
        BNE     FILBYT          ;COUNT BYTES
```

Chapter 5 contains a more general version.

Here again we can simplify the program by letting memory locations PGZRO and PGZRO+1 contain an address one less than the lowest address in the area to be filled. The revised program is

```
FILBYT  STA     (PGZRO),Y       ;FILL A BYTE
        DEY
        BNE     FILBYT          ;COUNT BYTES
```

Exchange Instructions

1. Exchange index registers X and Y.

```
STX     TEMP        ;SAVE X
TYA                 ;Y TO X
TAX
LDY     TEMP        ;X TO Y
```

or

```
TXA                 ;SAVE X
PHA
TYA                 ;Y TO X
TAX
PLA                 ;X TO Y
TAY
```

Both versions take the same number of bytes (assuming TEMP is on page 0). The second version is slower but reentrant.

2. Exchange memory locations ADDR1 and ADDR2.

```
LDA     ADDR1
LDX     ADDR2
STX     ADDR1
STA     ADDR2
```

3. Exchange accumulator and top of stack.

```
TAY                 ;SAVE A
PLA                 ;GET TOP OF STACK
TAX                 ;SAVE TOP OF STACK
TYA                 ;A TO TOP OF STACK
PHA
TXA                 ;TOP OF STACK TO A
```

Clear Instructions

1. Clear the accumulator.

```
LDA     #0
```

The 6502 treats 0 like any other number. There are no special clear instructions.

2. Clear an index register.

```
LDX     #0
```

or

```
LDY     #0
```

3. Clear memory location ADDR.

```
LDA     #0
STA     ADDR
```

Obviously, we could use X or Y as easily as A.

4. Clear memory locations ADDR and ADDR+1.

```
LDA     #0
STA     ADDR
STA     ADDR+1
```

5. Clear bit of accumulator.

```
AND     #MASK          ;CLEAR BIT BY MASKING
```

MASK has 0 bits in the positions to be cleared and 1 bits in the positions that are to be left unchanged. For example,

```
AND     #%10111110   ;CLEAR BITS 0 AND 6 OF A
```

Logically ANDing a bit with 1 leaves it unchanged.

Set Instructions

1. Set the accumulator to FF_{16} (all ones in binary).

```
LDA     #$FF
```

2. Set an index register to FF_{16}.

```
LDX     #$FF
```

or

```
LDY     #$FF
```

3. Set the stack pointer to FF_{16}.

```
LDX     #$FF
TXS
```

The next available location in the stack is at address $01FF_{16}$.

4. Set a memory location to FF_{16}.

```
LDA     #$FF
STA     ADDR
```

5. Set bit of accumulator.

```
ORA     #MASK          ;SET BIT BY MASKING
```

MASK has 1 bits in the positions to be set and 0 bits elsewhere. For example,

```
ORA     #%10000000   ;SET BIT 7 (SIGN BIT)
```

Logically ORing a bit with 0 leaves it unchanged.

BRANCH (JUMP) INSTRUCTIONS

Unconditional Branch Instructions

1. Unconditional branch relative to DEST.

```
CLC                     ;DELIBERATELY CLEAR CARRY
BCC     DEST            ;FORCE AN UNCONDITIONAL BRANCH
```

You can always force an unconditional branch by branching conditionally on a condition that is known to be true. Some obvious alternatives are

```
SEC
BCS     DEST
```

or

```
LDA     #0
BEQ     DEST
```

or

```
LDA     #1
BNE     DEST
```

2. Jump indirect to address at the top of the stack.

```
RTS
```

RTS is just an ordinary indirect jump in which the processor obtains the destination from the top of the stack. Be careful, however, of the fact that the processor adds 1 to the address before proceeding.

3. Jump indexed, assuming that the base of the address table is BASE and the index is in memory location INDEX. The addresses are arranged in the usual 6502 manner with the less significant byte first.[5]

- Using indirect addressing:

```
LDA     INDEX
ASL     A               ;DOUBLE INDEX FOR 2-BYTE ENTRIES
TAX
LDA     BASE,X          ;GET LSB OF DESTINATION
STA     INDIR
INX
LDA     BASE,X          ;GET MSB OF DESTINATION
STA     INDIR+1
JMP     (INDIR)         ;JUMP INDIRECT TO DESTINATION
```

- Using the stack:

```
LDA     INDEX
ASL     A               ;DOUBLE INDEX FOR 2-BYTE ENTRIES
TAX
LDA     BASE+1,X        ;GET MSB OF DESTINATION
PHA
```

```
LDA     BASE,X          ;GET LSB OF DESTINATION
PHA
RTS                     ;JUMP INDIRECT TO DESTINATION OFFSET 1
```

The second approach is faster but less straightforward. Note the following:

1. You must store the more significant byte first since the stack is growing toward lower addresses. Thus the bytes end up in their usual order.

2. Since RTS adds 1 to the program counter after loading it from the stack, the table entries must all be 1 less than the actual destination addresses for this method to work correctly.

3. Documentation is essential, since this method uses RTS for the rather surprising purpose of transferring control to a subroutine, rather than from it. The mnemonic may confuse the reader, but it obviously does not bother the microprocessor.

Conditional Branch Instructions

1. Branch if zero.

· Branch if accumulator contains zero.

```
TAX                     ;TEST ACCUMULATOR
BEQ     DEST
```

or

```
CMP     #0              ;TEST ACCUMULATOR
BEQ     DEST
```

Either AND #$FF or ORA #0 will set the Zero flag if (A)=0 without affecting the Carry flag (CMP#0 sets Carry).

· Branch if an index register contains 0.

```
CPX     #0              ;TEST INDEX REGISTER
BEQ     DEST
```

The instruction TXA or the sequence INX, DEX can be used to test the contents of index register X without affecting the Carry flag (CPX#0 sets the Carry). TXA, of course, changes the accumulator.

· Branch if a memory location contains 0.

```
INC     ADDR            ;TEST MEMORY LOCATION
DEC     ADDR
BEQ     DEST
```

or

```
LDA     ADDR            ;TEST MEMORY LOCATION
BEQ     DEST
```

· Branch if a pair of memory locations (ADDR and ADDR+1) both contain 0.

```
LDA     ADDR          ;TEST 16-BIT NUMBER FOR ZERO
ORA     ADDR+1
BEQ     DEST
```

· Branch if a bit of the accumulator is zero.

```
AND     #MASK         ;TEST BIT OF ACCUMULATOR
BEQ     DEST
```

MASK has a 1 bit in the position to be tested and 0s elsewhere. Note the inversion here; if the bit of the accumulator is a 0, the result is 0 and the Zero flag is set to 1. Special cases are

Bit position 7

```
ASL     A             ;MOVE BIT 7 TO CARRY
BCC     DEST
```

Bit position 6

```
ASL     A             ;MOVE BIT 6 TO NEGATIVE FLAG
BPL     DEST
```

Bit position 0

```
LSR     A             ;MOVE BIT 0 TO CARRY
BCC     DEST
```

· Branch if a bit of a memory location is 0.

```
LDA     #MASK
BIT     ADDR          ;TEST BIT OF MEMORY
BEQ     DEST
```

MASK has a 1 bit in the position to be tested and 0s elsewhere. Special cases are

Bit position 7

```
BIT     ADDR          ;TEST MEMORY
BPL     DEST          ;BRANCH ON BIT 7
```

Bit position 6

```
BIT     ADDR          ;TEST MEMORY
BVC     DEST          ;BRANCH ON BIT 6
```

The BIT instruction sets the Negative flag from bit 7 of the memory location and the Overflow flag from bit 6, regardless of the contents of the accumulator.

We can also use the shift instructions to test the bits at the ends, as long as we can tolerate changes in the memory locations.

Bit position 7

```
ASL     ADDR          ;TEST BIT 7
BCC     DEST
```

Bit position 6

```
ASL     ADDR          ;TEST BIT 6
BPL     DEST
```

Bit position 0

```
LSR     ADDR            ;TEST BIT 0
BCC     DEST
```

· Branch if the Interrupt Disable flag (bit 2 of the status register) is 0.

```
PHP                     ;MOVE STATUS TO A
PLA
AND     #%00000100      ;TEST INTERRUPT DISABLE
BEQ     DEST            ;BRANCH IF INTERRUPTS ARE ON
```

· Branch if the Decimal Mode flag (bit 3 of the status register) is 0.

```
PHP                     ;MOVE STATUS TO A
PLA
AND     #%00001000      ;TEST DECIMAL MODE FLAG
BEQ     DEST            ;BRANCH IF MODE IS BINARY
```

2. Branch if not 0.

· Branch if accumulator does not contain 0.

```
TAX                     ;TEST ACCUMULATOR
BNE     DEST
```

or

```
CMP     #0              ;TEST ACCUMULATOR
BNE     DEST
```

· Branch if an index register does not contain 0.

```
CPX     #0              ;TEST INDEX REGISTER
BNE     DEST
```

· Branch if a memory location does not contain 0.

```
INC     ADDR            ;TEST MEMORY LOCATION
DEC     ADDR
BNE     DEST
```

or

```
LDA     ADDR            ;TEST MEMORY LOCATION
BNE     DEST
```

· Branch if a pair of memory locations (ADDR and ADDR+1) do not both contain 0.

```
LDA     ADDR            ;TEST 16-BIT NUMBER FOR ZERO
ORA     ADDR+1
BNE     DEST
```

· Branch if a bit of the accumulator is 1.

```
AND     #MASK           ;TEST BIT OF ACCUMULATOR
BNE     DEST
```

MASK has a 1 bit in the position to be tested and 0s elsewhere. Note the inversion here; if the bit of the accumulator is a 1, the result is not 0 and the Zero flag is set to 0. Special cases are

Bit position 7

```
ASL     A           ;MOVE BIT 7 TO CARRY
BCS     DEST        ;AND TEST CARRY
```

Bit position 6

```
ASL     A           ;MOVE BIT 6 TO SIGN
BMI     DEST        ;AND TEST SIGN
```

Bit position 0

```
LSR     A           ;MOVE BIT 0 TO CARRY
BCS     DEST        ;AND TEST CARRY
```

- Branch if a bit of a memory location is 1.

```
LDA     #MASK
BIT     ADDR        ;TEST BIT OF MEMORY
BNE     DEST
```

MASK has a 1 bit in the position to be tested and 0s elsewhere. Special cases are

Bit position 7

```
BIT     ADDR        ;TEST BIT 7 OF MEMORY
BMI     DEST
```

Bit position 6

```
BIT     ADDR        ;TEST BIT 6 OF MEMORY
BVS     DEST
```

The BIT instruction sets the Negative flag from bit 7 of the memory location and the Overflow flag from bit 6, regardless of the contents of the accumulator.

We can also use the shift instructions to test the bits at the ends, as long as we can tolerate changes in the memory locations.

Bit position 7

```
ASL     ADDR        ;TEST BIT 7 OF MEMORY
BCS     DEST
```

This alternative is slower than BIT by 2 clock cycles, since it must write the result back into memory.

Bit position 6

```
ASL     ADDR        ;TEST BIT 6 OF MEMORY
BMI     DEST
```

Bit position 0

```
LSR     ADDR        ;TEST BIT 0 OF MEMORY
BCS     DEST
```

· Branch if the Interrupt Disable flag (bit 2 of the status register) is 1.

```
        PHP                 ;MOVE STATUS TO A THROUGH STACK
        PLA
        AND     #%00000100  ;TEST INTERRUPT DISABLE
        BNE     DEST        ;BRANCH IF INTERRUPTS ARE DISABLED
```

· Branch if the Decimal Mode flag (bit 3 of the status register) is 1.

```
        PHP                 ;MOVE STATUS TO A THROUGH STACK
        PLA
        AND     #%00001000  ;TEST DECIMAL MODE FLAG
        BNE     DEST        ;BRANCH IF MODE IS DECIMAL
```

3. Branch if Equal.

· Branch if (A) = VALUE.

```
        CMP     #VALUE      ;COMPARE BY SUBTRACTING
        BEQ     DEST
```

· Branch if (X) = VALUE.

```
        CPX     #VALUE      ;COMPARE BY SUBTRACTING
        BEQ     DEST
```

Two special cases are

Branch if (X) = 1

```
        DEX
        BEQ     DEST
```

Branch if (X) = FF_{16}.

```
        INX
        BEQ     DEST
```

· Branch if (A) = (ADDR).

```
        CMP     ADDR        ;COMPARE BY SUBTRACTING
        BEQ     DEST
```

· Branch if (X) = (ADDR).

```
        CPX     ADDR        ;COMPARE BY SUBTRACTING
        BEQ     DEST
```

. Branch if the contents of memory locations PGZRO and PGZRO + 1 equal VAL16 (VAL16L less significant byte, VAL16M more significant byte).

```
        LDA     PGZRO+1     ;COMPARE MSB'S
        CMP     #VAL16M
        BNE     DONE
        LDA     PGZRO       ;AND LSB'S ONLY IF NECESSARY
        CMP     #VAL16L
        BEQ     DEST
DONE    NOP
```

· Branch if the contents of memory locations PGZRO and PGZRO + 1 equal those of memory locations LIML and LIMH.

```
        LDA     PGZRO+1         ;COMPARE MSB'S
        CMP     LIMH
        BNE     DONE
        LDA     PGZRO           ;AND LSB'S ONLY IF NECESSARY
        CMP     LIML
        BEQ     DEST
DONE    NOP
```

Note: Neither of the next two sequences should be used to test for stack overflow or underflow, since intervening instructions (for example, a single JSR or RTS) could change the stack pointer by more than 1.

- Branch if (S) = VALUE.

```
        TSX                     ;CHECK IF STACK IS AT LIMIT
        CPX     #VALUE
        BEQ     DEST
```

- Branch if (S) = (ADDR).

```
        TSX                     ;CHECK IF STACK IS AT LIMIT
        CPX     ADDR
        BEQ     DEST
```

4. Branch if Not Equal.
   - Branch if (A) $\neq$ VALUE.

```
        CMP     #VALUE          ;COMPARE BY SUBTRACTING
        BNE     DEST
```

   - Branch if (X) $\neq$ VALUE.

```
        CPX     #VALUE          ;COMPARE BY SUBTRACTING
        BNE     DEST
```

   Two special cases are
   Branch if (X) $\neq$ 1.

```
        DEX
        BNE     DEST
```

   - Branch if (X) $\neq FF_{16}$.

```
        INX
        BNE     DEST
```

   - Branch if (A) $\neq$ (ADDR).

```
        CMP     ADDR            ;COMPARE BY SUBTRACTING
        BNE     DEST
```

   - Branch if (X) $\neq$ (ADDR).

```
        CPX     ADDR            ;COMPARE BY SUBTRACTING
        BNE     DEST
```

- Branch if the contents of memory locations PGZRO and PGZRO+1 are not equal to VAL16 (VAL16L less significant byte, VAL16M more significant byte).

```
LDA     PGZRO+1         ;COMPARE MSB'S
CMP     #VAL16M
BNE     DEST
LDA     PGZRO           ;AND LSB'S ONLY IF NECESSARY
CMP     #VAL16L
BNE     DEST
```

· Branch if the contents of memory locations PGZRO and PGZRO + 1 are not equal to those of memory locations LIML and LIMH.

```
LDA     PGZRO+1         ;COMPARE MSB'S
CMP     LIMH
BNE     DEST
LDA     PGZRO           ;COMPARE LSB'S ONLY IF NECESSARY
CMP     LIML
BNE     DEST
```

Note: Neither of the next two sequences should be used to test for stack overflow or underflow, since intervening instructions (for example, a single JSR or RTS) could change the stack pointer by more than 1.

· Branch if (S) ≠ VALUE.

```
TSX                     ;CHECK IF STACK IS AT LIMIT
CPX     #VALUE
BNE     DEST
```

· Branch if (S) ≠ (ADDR).

```
TSX                     ;CHECK IF STACK IS AT LIMIT
CPX     ADDR
BNE     DEST
```

5. Branch if Positive.

· Branch if contents of accumulator are positive.

```
TAX                     ;TEST ACCUMULATOR
BPL     DEST
```

or

```
CMP     #0              ;TEST ACCUMULATOR
BPL     DEST
```

· Branch if contents of index register X are positive.

```
TXA                     ;TEST REGISTER X
BPL     DEST
```

or

```
CPX     #0              ;TEST INDEX REGISTER X
BPL     DEST
```

· Branch if contents of a memory location are positive.

```
LDA     ADDR            ;TEST A MEMORY LOCATION
BPL     DEST
```

or

```
BIT     ADDR
BPL     DEST
```

· Branch if 16-bit number in memory locations ADDR and ADDR+1 (MSB in ADDR+1) is positive.

```
        BIT     ADDR+1          ;TEST MSB
        BPL     DEST
```

Remember that BIT sets the Negative flag from bit 7 of the memory location, regardless of the contents of the accumulator.

6. Branch if Negative.

· Branch if contents of accumulator are negative.

```
        TAX                     ;TEST ACCUMULATOR
        BMI     DEST
```

or

```
        CMP     #0              ;TEST ACCUMULATOR
        BMI     DEST
```

· Branch if contents of index register X are negative.

```
        TXA                     ;TEST REGISTER X
        BMI     DEST
```

or

```
        CPX     #0              ;TEST INDEX REGISTER X
        BMI     DEST
```

· Branch if contents of a memory location are negative.

```
        BIT     ADDR            ;TEST A MEMORY LOCATION
        BMI     DEST
```

or

```
        LDA     ADDR            ;TEST A MEMORY LOCATION
        BMI     DEST
```

· Branch if 16-bit number in memory locations ADDR and ADDR+1 (MSB in ADDR+1) is negative.

```
        BIT     ADDR+1          ;TEST MSB
        BMI     DEST
```

Remember that BIT sets the Negative flag from bit 7 of the memory location, regardless of the contents of the accumulator.

7. Branch if Greater Than (Signed).

· Branch if (A) > VALUE.

```
        CMP     #VALUE          ;COMPARE BY SUBTRACTING
        BEQ     DONE            ;NO BRANCH IF EQUAL
        BVS     CHKOPP          ;DID OVERFLOW OCCUR?
        BPL     DEST            ;NO, THEN BRANCH ON POSITIVE
        BMI     DONE
CHKOPP  BMI     DEST            ;YES, THEN BRANCH ON NEGATIVE
DONE    NOP
```

The idea here is to branch if the result is greater than zero and overflow did not occur, or if the result is less than zero and overflow did occur. Overflow makes the apparent sign the opposite of the real sign.

· Branch if (A) > (ADDR).

```
        CMP     ADDR        ;COMPARE BY SUBTRACTING
        BEQ     DONE        ;NO BRANCH IF EQUAL
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BPL     DEST        ;NO, THEN BRANCH ON POSITIVE
        BMI     DONE
CHKOPP  BMI     DEST        ;YES, THEN BRANCH ON NEGATIVE
DONE    NOP
```

8. Branch if Greater Than or Equal To (Signed)

· Branch if (A) ≥ VALUE.

```
        CMP     #VALUE      ;COMPARE BY SUBTRACTING
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BPL     DEST        ;NO, THEN BRANCH ON POSITIVE
        BMI     DONE
CHKOPP  BMI     DEST        ;YES, THEN BRANCH ON NEGATIVE
DONE    NOP
```

The idea here is to branch if the result is greater than or equal to 0 and overflow did not occur, or if the result is less than 0 and overflow did occur.

· Branch if (A) ≥ (ADDR).

```
        CMP     ADDR        ;COMPARE BY SUBTRACTING
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BPL     DEST        ;NO, THEN BRANCH ON POSITIVE
        BMI     DONE
CHKOPP  BMI     DEST        ;YES, THEN BRANCH ON NEGATIVE
DONE    NOP
```

9. Branch if Less Than (Signed)

· Branch if (A) < VALUE (signed).

```
        CMP     #VALUE      ;COMPARE BY SUBTRACTING
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BMI     DEST        ;NO, THEN BRANCH ON NEGATIVE
        BPL     DONE
CHKOPP  BPL     DEST        ;YES, THEN BRANCH ON POSITIVE
DONE    NOP
```

The idea here is to branch if the result is negative and overflow did not occur, or if the result is positive but overflow did occur.

· Branch if (A) < (ADDR) (signed).

```
        CMP     ADDR        ;COMPARE BY SUBTRACTING
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BMI     DEST        ;NO, THEN BRANCH ON NEGATIVE
        BPL     DONE
CHKOPP  BPL     DEST        ;YES, THEN BRANCH ON POSITIVE
DONE    NOP
```

10. Branch if Less Than or Equal (Signed).

· Branch if (A) $\leq$ VALUE (signed).

```
        CMP     #VALUE      ;COMPARE BY SUBTRACTING
        BEQ     DEST        ;BRANCH IF EQUAL
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BMI     DEST        ;NO, THEN BRANCH ON NEGATIVE
        BPL     DONE
CHKOPP  BPL     DEST        ;YES, THEN BRANCH ON POSITIVE
DONE    NOP
```

The idea here is to branch if the result is 0, negative without overflow, or positive with overflow.

· Branch if (A) $\leq$ (ADDR) (signed).

```
        CMP     ADDR        ;COMPARE BY SUBTRACTING
        BEQ     DEST        ;BRANCH IF EQUAL
        BVS     CHKOPP      ;DID OVERFLOW OCCUR?
        BMI     DEST        ;NO, THEN BRANCH ON NEGATIVE
        BPL     DONE
CHKOPP  BPL     DEST        ;YES, THEN BRANCH ON POSITIVE
DONE    NOP
```

11. Branch if Higher (Unsigned). That is, branch if the unsigned comparison is nonzero and does not require a borrow.

· Branch if (A) > VALUE (unsigned).

```
        CMP     #VALUE      ;COMPARE BY SUBTRACTING
        BEQ     DONE        ;NO BRANCH IF EQUAL
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
DONE    NOP
```

or

```
        CMP     #VALUE+1    ;COMPARE BY SUBTRACTING VALUE + 1
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
```

It is shorter and somewhat more efficient to simply compare to a number one higher than the actual threshold. Then we can use BCS, which causes a branch if the contents of the accumulator are greater than or equal to VALUE+1 (unsigned).

· Branch if (A) > (ADDR) (unsigned).

```
        CMP     ADDR        ;COMPARE BY SUBTRACTING
        BEQ     DONE        ;NO BRANCH IF EQUAL
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
DONE    NOP
```

· Branch if (X) > VALUE (unsigned).

```
        CPX     #VALUE+1    ;COMPARE BY SUBTRACTING VALUE+1
        BCS     DEST
```

· Branch if (X) > (ADDR) (unsigned).

```
        CPX     ADDR        ;COMPARE BY SUBTRACTING
        BEQ     DONE        ;NO BRANCH IF EQUAL
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
DONE    NOP
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are larger (unsigned) than VAL16 (VAL16L less significant byte, VAL16M more significant byte).

```
        LDA     #VAL16L     ;GENERATE BORROW BY COMPARING LSB'S
        CMP     PGZRO
        LDA     #VAL16M     ;COMPARE MSB'S WITH BORROW
        SBC     PGZRO+1
        BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are larger (unsigned) than the contents of memory locations LIML and LIMH (MSB in LIMH).

```
        LDA     LIML        ;GENERATE BORROW BY COMPARING LSB'S
        CMP     PGZRO
        LDA     LIMH        ;COMPARE MSB'S WITH BORROW
        SBC     PGZRO+1
        BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if (S) > VALUE (unsigned).

```
        TSX                 ;CHECK IF STACK BEYOND LIMIT
        CPX     #VALUE
        BEQ     DONE        ;NO BRANCH IF EQUAL
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
DONE    NOP
```

or

```
        TSX                 ;CHECK IF STACK BEYOND LIMIT
        CPX     #VALUE+1    ;COMPARE BY SUBTRACTING VALUE + 1
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
```

· Branch if (S) > (ADDR) (unsigned).

```
        TSX                 ;CHECK IF STACK BEYOND LIMIT
        BEQ     DONE        ;NO BRANCH IF EQUAL
        BCS     DEST        ;BRANCH IF NO BORROW NEEDED
DONE    NOP
```

12. Branch if Not Higher (Unsigned). Branch if the unsigned comparison is 0 or requires a borrow.

Branch if (A) $\leq$ VALUE (unsigned).

```
        CMP     #VALUE      ;COMPARE BY SUBTRACTING
        BCC     DEST        ;BRANCH IF BORROW NEEDED
        BEQ     DEST        ;BRANCH IF EQUAL
```

If the two values are the same, CMP sets the Carry to indicate that no borrow was necessary.

or

```
CMP     #VALUE+1      ;COMPARE BY SUBTRACTING VALUE + 1
BCC     DEST          ;BRANCH IF BORROW NEEDED
```

· Branch if (A) $\leq$ (ADDR) (unsigned).

```
CMP     ADDR          ;COMPARE BY SUBTRACTING
BCC     DEST          ;BRANCH IF BORROW NEEDED
BEQ     DEST          ;BRANCH IF EQUAL
```

· Branch if (X) $\leq$ VALUE (unsigned).

```
CPX     #VALUE        ;COMPARE BY SUBTRACTING
BCC     DEST          ;BRANCH IF BORROW NEEDED
BEQ     DEST          ;BRANCH IF EQUAL
```

or

```
CPX     #VALUE+1      ;COMPARE BY SUBTRACTING VALUE + 1
BCC     DEST          ;BRANCH IF BORROW NEEDED
```

· Branch if (X) $\leq$ (ADDR) (unsigned).

```
CPX     ADDR          ;COMPARE BY SUBTRACTING
BCC     DEST          ;BRANCH IF BORROW NEEDED
BEQ     DEST          ;BRANCH IF EQUAL
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are less than or equal to (unsigned) VAL16 (VAL16M more significant byte, VAL16L less significant byte).

```
LDA     #VAL16L       ;GENERATE BORROW BY COMPARING LSB'S
CMP     PGZRO
LDA     #VAL16M       ;COMPARE MSB'S WITH BORROW
SBC     PGZRO+1
BCS     DEST          ;BRANCH IF NO BORROW GENERATED
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are less than or equal to (unsigned) the contents of memory locations LIML and LIMH (MSB in LIMH).

```
LDA     LIML          ;GENERATE BORROW BY COMPARING LSB'S
CMP     PGZRO
LDA     LIMH          ;COMPARE MSB'S WITH BORROW
SBC     PGZRO+1
BCS     DEST          ;BRANCH IF NO BORROW GENERATED
```

· Branch if (S) $\leq$ VALUE (unsigned).

```
TSX                   ;CHECK IF STACK AT OR BELOW LIMIT
CPX     #VALUE
BCC     DEST          ;BRANCH IF BORROW NEEDED
BEQ     DEST          ;BRANCH IF EQUAL
```

or

```
TSX                 ;CHECK IF STACK AT OR BELOW LIMIT
CPX     #VALUE+1    ;COMPARE BY SUBTRACTING VALUE + 1
BCC     DEST
```

· Branch if (S) ≤ (ADDR) (unsigned).

```
TSX                 ;CHECK IF STACK AT OR BELOW LIMIT
CPX     ADDR
BCC     DEST        ;BRANCH IF BORROW NEEDED
BEQ     DEST        ;BRANCH IF EQUAL
```

13. Branch if Lower (Unsigned). That is, branch if the unsigned comparison requires a borrow.

· Branch if (A) < (unsigned).

```
CMP     #VALUE      ;COMPARE BY SUBTRACTING
BCC     DEST        ;BRANCH IF BORROW GENERATED
```

The Carry flag is set to 0 if the subtraction generates a borrow.

· Branch if (A) < (ADDR) (unsigned).

```
CMP     ADDR        ;COMPARE BY SUBTRACTING
BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if (X) < VALUE (unsigned).

```
CPX     #VALUE      ;COMPARE BY SUBTRACTING
BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if (X) < (ADDR) (unsigned).

```
CPX     ADDR        ;COMPARE BY SUBTRACTING
BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are less than (unsigned) VAL16 (VAL16L less significant byte, VAL16M more significant byte).

```
LDA     PGZRO       ;GENERATE BORROW BY COMPARING LSB'S
CMP     #VAL16L
LDA     PGZRO+1     ;COMPARE MSB'S WITH BORROW
SBC     #VAL16M
BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are less than (unsigned) the contents of memory locations LIML and LIMH (MSB in LIMH).

```
LDA     PGZRO       ;GENERATE BORROW BY COMPARING LSB'S
CMP     LIML
LDA     PGZRO+1     ;COMPARE MSB'S WITH BORROW
SBC     LIMH
BCC     DEST        ;BRANCH IF BORROW GENERATED
```

· Branch if (S) < VALUE (unsigned).

```
TSX                 ;CHECK IF STACK BELOW LIMIT
CPX     #VALUE
BCC     DEST        ;BRANCH IF BORROW NEEDED
```

· Branch if (S) < (ADDR) (unsigned).

```
TSX                 ;CHECK IF STACK BELOW LIMIT
CPX     ADDR
BCC     DEST        ;BRANCH IF BORROW NEEDED
```

14. Branch if Not Lower (Unsigned). That is, branch if the unsigned comparison does not require a borrow.

· Branch if (A) ≥ VALUE (unsigned).

```
CMP     #VALUE      ;COMPARE BY SUBTRACTING
BCS     DEST        ;BRANCH IF NO BORROW GENERATED
```

The Carry flag is set to one if the subtraction does not generate a borrow.

· Branch if (A) ≥ (ADDR) (unsigned).

```
CMP     ADDR        ;COMPARE BY SUBTRACTING
BCS     DEST
```

· Branch if (X) ≥ VALUE (unsigned).

```
CPX     #VALUE      ;COMPARE BY SUBTRACTING
BCS     DEST        ;BRANCH IF NO BORROW GENERATED
```

· Branch if (X) ≥ (ADDR) (unsigned).

```
CPX     ADDR        ;COMPARE BY SUBTRACTING
BCS     DEST
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are greater than or equal to (unsigned) VAL16 (VAL16L less significant byte, VAL16M more significant byte).

```
LDA     PGZRO       ;GENERATE BORROW BY COMPARING LSB'S
CMP     #VAL16L
LDA     PGZRO+1     ;COMPARE MSB'S WITH BORROW
SBC     #VAL16M
BCS     DEST        ;BRANCH IF NO BORROW GENERATED
```

· Branch if the contents of memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1) are greater than or equal to (unsigned) the contents of memory locations LIML and LIMH (MSB in LIMH).

```
LDA     PGZRO       ;GENERATE BORROW BY COMPARING LSB'S
CMP     LIML
LDA     PGZRO+1     ;COMPARE MSB'S WITH BORROW
SBC     LIMH
BCS     DEST        ;BRANCH IF NO BORROW GENERATED
```

· Branch if (S) $\geq$ VALUE (unsigned).

```
TSX                 ;CHECK IF STACK AT OR ABOVE LIMIT
CPX     #VALUE
BCS     DEST        ;BRANCH IF NO BORROW NEEDED
```

· Branch if (S) $\geq$ (ADDR) (unsigned).

```
TSX                 ;CHECK IF STACK AT OR ABOVE LIMIT
CPX     ADDR
BCS     DEST        ;BRANCH IF NO BORROW NEEDED
```

SKIP INSTRUCTIONS

You can implement skip instructions on the 6502 microprocessor by using branch or jump instructions with the proper destination. That destination should be one instruction beyond the one that the processor would execute sequentially after the branch. Note that skip instructions are awkward to implement on most microprocessors, because their instructions vary in length and it is difficult to determine how long a jump is required to skip an instruction.

SUBROUTINE CALL INSTRUCTIONS

Unconditional Call Instructions

You can implement an indirect call on the 6502 microprocessor by calling a routine that performs an ordinary indirect jump. A RETURN FROM SUBROUTINE (RTS) instruction at the end of the subroutine will then transfer control back to the original calling point. The main program performs

```
JSR     TRANS
```

where TRANS is the subroutine that actually transfers control using a jump instruction. Note that TRANS ends with a jump, not with a return. Typical TRANS routines are:

· To address in memory locations INDIR and INDIR+1 (MSB in INDIR+1).

```
JMP     (INDIR)
```

· To address in table starting at memory location BASE and using index in memory location INDEX.

```
        LDA     INDEX
        ASL     A               ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        TAX
        LDA     BASE,X          ;GET LSB OF DESTINATION
        STA     INDIR
        INX
        LDA     BASE,X          ;GET MSB OF DESTINATION
        STA     INDIR+1
        JMP     (INDIR)         ;JUMP INDIRECT TO DESTINATION
```

or

```
        LDA     INDEX
        ASL     A               ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        TAX
        LDA     BASE+1,X        ;GET MSB OF DESTINATION
        PHA
        LDA     BASE,X          ;GET LSB OF DESTINATION
        PHA
        RTS                     ;JUMP TO DESTINATION PLUS 1
```

In the second approach, the table must contain the actual destination addresses minus 1, since RTS adds 1 to the program counter after loading it from the stack.

Conditional Call Instructions

You can implement a conditional call on the 6502 microprocessor by branching on the opposite condition around the call. For example, you could provide CALL ON CARRY CLEAR with the sequence

```
        BCS     NEXT            ;BRANCH AROUND IF CARRY SET
        JSR     SUBR            ;CALL IF CARRY CLEAR
NEXT    NOP
```

RETURN INSTRUCTIONS

Unconditional Return Instructions

The RTS instruction returns control automatically to the address saved at the top of the stack (plus 1). If the return address is saved elsewhere (i.e., in two memory locations), you can return control to it by performing an indirect jump. Note that you must add 1 to the return address to simulate RTS.

The following sequence pops the return address from the top of the stack, adds 1 to it, and stores the adjusted value in memory locations RETADR and RETADR+1.

```
        PLA                 ;POP LSB OF RETURN ADDRESS
        CLC                 ;ADD 1 TO LSB
        ADC     #1
        STA     RETADR
        PLA                 ;POP MSB OF RETURN ADDRESS
        ADC     #0          ;ADD CARRY TO MSB
        STA     RETADR+1
```

A final JMP (RETADR) will now transfer control to the proper place.

Conditional Return Instructions

You can implement conditional returns on the 6502 microprocessor by using the conditional branches (on the opposite condition) to branch around an RTS instruction. That is, for example, you could provide RETURN ON NOT ZERO with the sequence

```
        BEQ     NEXT        ;BRANCH AROUND IF ZERO
        RTS                 ;RETURN ON NOT ZERO
NEXT    NOP
```

Return with Skip Instructions

· Return control to the address at the top of the stack after it has been incremented by an offset NUM. This sequence allows you to transfer control past parameters, data, or other nonexecutable items.

```
        PLA                 ;POP RETURN ADDRESS
        CLC
        ADC     #NUM+1      ;INCREMENT BY NUM
        STA     RETADR
        PLA
        ADC     #0          ;WITH CARRY IF NECESSARY
        STA     RETADR+1
        JMP     (RETADR)
```

or

```
        TSX                 ;MOVE STACK POINTER TO INDEX REGISTER
        LDA     $0101,X     ;INCREMENT RETURN ADDRESS BY NUM
        CLC
        ADC     #NUM
        STA     $0101,X
        BCC     DONE
        INC     $0102,X     ;WITH CARRY IF NECESSARY
DONE    RTS
```

· Change the return address to RETPT. Assume that the return address is stored currently at the top of the stack. RETPT consists of RETPTH (MSB) and RETPTL (LSB).

```
TSX
LDA     #RETPTL
STA     $0101,X
LDA     #RETPT
STA     $0102,X
RTS
```

The actual return point is RETPT + 1.

Return from Interrupt Instructions

If the initial portion of the interrupt service routine saves all the registers with the sequence.

```
PHA                 ;SAVE ACCUMULATOR
TXA                 ;SAVE INDEX REGISTER X
PHA
TYA                 ;SAVE INDEX REGISTER Y
PHA
```

A standard return sequence is

```
PLA                 ;RESTORE INDEX REGISTER Y
TAY
PLA                 ;RESTORE INDEX REGISTER X
TAX
PLA                 ;RESTORE ACCUMULATOR
```

MISCELLANEOUS INSTRUCTIONS

In this category, we include push and pop instructions, halt, wait, break, decimal adjust, enabling and disabling of interrupts, translation (table lookup), and other instructions that do not fall into any of the earlier categories.

1. Push Instructions.

· Push index register X.

```
TXA                 ;SAVE X IN STACK VIA A
PHA
```

· Push index register Y.

```
TYA                 ;SAVE Y IN STACK VIA A
PHA
```

· Push memory location ADDR.

```
LDA     ADDR        ;SAVE MEMORY LOCATION IN STACK
PHA
```

ADDR could actually be an external priority register or a copy of it.

· Push memory locations ADDR and ADDR+1 (ADDR+1 most significant).

```
LDA     ADDR+1        ;SAVE 16-BIT NUMBER IN STACK
PHA
LDA     ADDR
PHA
```

Since the stack is growing toward lower addresses, the 16-bit number ends up stored in its usual 6502 form.

2. Pop (pull) instructions.

· Pop index register X.

```
PLA                   ;RESTORE X FROM STACK VIA A
TAX
```

· Pop index register Y.

```
PLA                   ;RESTORE Y FROM STACK VIA A
TAY
```

· Pop memory location ADDR.

```
PLA                   ;RESTORE MEMORY LOCATION FROM STACK
STA     ADDR
```

ADDR could actually be an external priority register or a copy of it.

· Pop memory locations ADDR and ADDR+1 (ADDR+1 most significant byte).

```
PLA                   ;RESTORE 16-BIT NUMBER FROM STACK
STA     ADDR
PLA
STA     ADDR+1
```

We assume that the 16-bit number is stored in the usual 6502 form with the less significant byte at the lower address.

Wait Instructions

The simplest way to implement a wait on the 6502 microprocessor is to use an endless loop such as:

```
HERE    JMP     HERE
```

The processor will continue executing the instruction until it is interrupted and will resume executing it after the interrupt service routine returns control. Of course, maskable interrupts must have been enabled or the processor will

execute the loop endlessly. The nonmaskable interrupt can interrupt the processor at any time.

Another alternative is a sequence that waits for a high-to-low transition on the Set Overflow input. Such a transition sets the Overflow (V) flag. So the required sequence is

```
        CLV                 ;CLEAR THE OVERFLOW FLAG
WAIT    BVC     WAIT        ;AND WAIT FOR A TRANSITION TO SET IT
```

This sequence is essentially a "Wait for Input Transition" instruction.

Adjust Instructions

1. Branch if accumulator does not contain a valid decimal (BCD) number.

```
        STA     TEMP        ;SAVE ACCUMULATOR
        SED                 ;ENTER DECIMAL MODE
        CLC                 ;ADD 0 IN DECIMAL MODE
        ADC     #0
        CLD                 ;LEAVE DECIMAL MODE
```

2. Decimal increment accumulator (add 1 to A in decimal).

```
        SED                 ;ENTER DECIMAL MODE
        CLC
        ADC     #1          ;ADD 1 DECIMAL
        CLD                 ;LEAVE DECIMAL MODE
```

3. Decimal decrement accumulator (subtract 1 from A in decimal).

```
        SED                 ;ENTER DECIMAL MODE
        SEC
        SBC     #1          ;SUBTRACT 1 DECIMAL
        CLD                 ;LEAVE DECIMAL MODE
```

4. Enter decimal mode but save the old Decimal Mode flag.

```
        PHP                 ;SAVE OLD DECIMAL MODE FLAG
        SED                 ;ENTER DECIMAL MODE
```

A final PLP instruction will restore the old value of the Decimal Mode flag (and the rest of the status register as well).

5. Enter binary mode but save the old Decimal Mode flag.

```
        PHP                 ;SAVE OLD DECIMAL MODE FLAG
        CLD                 ;ENTER BINARY MODE
```

A final PLP instruction will restore the old value of the Decimal Mode flag (and the rest of the status register as well).

Enable and Disable Interrupt Instructions

1. Enable interrupts but save previous value of I flag.

```
PHP                 ;SAVE OLD I FLAG
CLI                 ;ENABLE INTERRUPTS
```

After a sequence that must run with interrupts enabled, a PLP instruction will restore the previous state of the interrupt system (and the rest of the status register as well).

2. Disable interrupts but save previous value of I flag.

```
PHP                 ;SAVE OLD I FLAG
SEI                 ;DISABLE INTERRUPTS
```

After a sequence that must run with interrupts disabled, a PLP instruction will restore the previous state of the interrupt system (and the rest of the status register as well).

Translate Instructions

1. Translate the operand in A to a value obtained from the corresponding entry in a table starting at the address in memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1).

```
TAY
LDA     (PGZRO),Y   ;REPLACE OPERAND WITH TABLE ENTRY
```

This procedure can be used to convert data from one code to another.

2. Translate the operand in A to a 16-bit value obtained from the corresponding entry in a table starting at the address in memory locations PGZRO and PGZRO+1 (MSB in PGZRO+1). Store the entry in memory locations TEMPL and TEMPH (MSB in TEMPH).

```
ASL     A           ;DOUBLE INDEX FOR 2-BYTE ENTRIES
TAY
LDA     (PGZRO),Y   ;GET LSB OF ENTRY
STA     TEMPL
INY
LDA     (PGZRO),Y   ;GET MSB OF ENTRY
STA     TEMPH
```

ADDITIONAL ADDRESSING MODES

· **Indirect Addressing**. You can provide indirect addressing on the 6502 processor (for addresses on page 0) by using the postindexed (indirect indexed)

addressing mode with register Y set to 0. A somewhat less powerful alternative (because you cannot index from the indirect address) is to use preindexing (indexed indirect addressing) with register X set to 0. Otherwise, indirect addressing is available only for the JMP instruction. Note that with JMP, the indirect address may be located anywhere in memory; it is not restricted to page 0.

Examples

1. Load the accumulator indirectly from the address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;SET INDEX TO ZERO
LDA     (PGZRO),Y       ;LOAD INDIRECT INDEXED
```

b. Store the accumulator indirectly at the address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;SET INDEX TO ZERO
STA     (PGZRO),Y       ;STORE INDIRECT INDEXED
```

In the case of instructions that lack the indirect indexed mode (such as ASL, DEC, INC, LSR, ROL, ROR), you must move the data to the accumulator, operate on it there, and then store it back in memory.

3. Increment the data at the address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;SET INDEX TO ZERO
LDA     (PGZRO),Y       ;GET THE DATA
CLC
ADC     #1              ;INCREMENT THE DATA
STA     (PGZRO),Y       ;STORE THE RESULT BACK
```

4. Logically shift right the data at the address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;SET INDEX TO ZERO
LDA     (PGZRO),Y       ;GET THE DATA
LSR     A               ;SHIFT IT RIGHT
STA     (PGZRO),Y       ;STORE THE RESULT BACK
```

5. Clear the address in memory locations PGZRO and PGZRO+1.

```
LDY     #0              ;SET INDEX TO ZERO
TYA                     ;DATA = ZERO
STA     (PGZRO),Y       ;CLEAR THE INDIRECT ADDRESS
```

The only way to provide indirect addressing for other pages is to move the indirect address to page 0 first.

6. Clear the address in memory locations INDIR and INDIR+1 (not on page 0).

```
LDA     INDIR          ;MOVE INDIRECT ADDRESS TO PAGE ZERO
STA     PGZRO
LDA     INDIR+1
STA     PGZRO+1
LDY     #0             ;SET INDEX TO ZERO
TYA                    ;DATA = ZERO
STA     (PGZRO),Y      ;CLEAR THE INDIRECT ADDRESS
```

· **Indexed Addressing**. Indexed addressing is available for most instructions in the 6502 set. We will discuss briefly the handling of the few for which it is not available and we will then discuss the handling of indexes that are larger than 256.

No indexing is available for BIT, CPX, CPY, JMP, and JSR. Only page 0 indexing is available for STX and STY. We can overcome these limitations as follows:

1. BIT

BIT indexed can be simulated by saving the accumulator, using AND, and restoring the accumulator. You should note that restoring the accumulator with LDA, PHA, TXA, or TYA will affect the Zero and Negative flags. A typical sequence without restoring the accumulator is:

```
PHA                    ;SAVE A
AND     BASE,X         ;LOGICAL AND INDEXED
```

The Zero flag is set as if an indexed BIT had been executed and the contents of A are available at the top of the stack.

2. CPX or CPY

CPX or CPY indexed can be simulated by moving the index register to A and using CMP. That is, CPX indexed with Y can be simulated by the sequence:

```
TXA                    ;MOVE X TO A
CMP     BASE,Y         ;THEN COMPARE INDEXED
```

3. JMP

JMP indexed can be simulated by calculating the required indexed address, storing it in memory, and using either JMP indirect or RTS to transfer control. The sequences are:

```
LDA     INDEX
ASL     A              ;DOUBLE INDEX FOR 2-BYTE ENTRIES
TAX
LDA     BASE,X         ;GET LSB OF DESTINATION
STA     INDIR
INX
LDA     BASE,X         ;GET MSB OF DESTINATION
STA     INDIR+1
JMP     (INDIR)        ;JUMP INDIRECT TO DESTINATION
```

or

```
LDA     INDEX
ASL     A               ;DOUBLE INDEX FOR 2-BYTE ENTRIES
TAX
LDA     BASE+1,X        ;GET MSB OF DESTINATION
PHA
LDA     BASE,X          ;GET LSB OF DESTINATION
PHA
RTS                     ;JUMP INDIRECT TO DESTINATION OFFSET 1
```

The second approach requires that the table contain entries that are all 1 less than the actual destinations, since RTS adds 1 to the program counter after restoring it from the stack.

4. JSR

JSR indexed can be simulated by calling a transfer program that executes JMP indexed as shown above. The ultimate return address remains at the top of the stack and a final RTS instruction will transfer control back to the original calling program. That is, the main program contains:

```
JSR     TRANS
```

TRANS performs an indexed jump and thus transfers control to the actual subroutine.

5. STX or STY

STX or STY indexed can be simulated by moving the index register to A and using STA. That is, we can simulate STX indexed with Y by using the sequence:

```
TXA                     ;MOVE X TO A
STA     BASE,Y          ;THEN STORE INDEXED
```

BASE can be anywhere in memory, not just on page 0.

We can handle indexes that are larger than 256 by performing an explicit addition on the more significant bytes and using the indirect indexed addressing mode. That is, if the base address is in memory locations PGZRO and PGZRO+1 and the index is in memory locations INDEX and INDEX+1, the following sequence will place the corrected base address in memory locations TEMP and TEMP+1 (on page 0).

```
LDA     PGZRO           ;SIMPLY MOVE LSB
STA     TEMP
LDA     PGZRO+1         ;ADD MSB'S
CLC
ADC     INDEX+1
STA     TEMP+1
```

TEMP and TEMP+1 now contain a base address that can be used (in conjunction with INDEX) in the indirect indexed mode.

Examples

1. Load accumulator indexed.

```
        LDY     INDEX           ;GET LSB OF INDEX
        LDA     (TEMP),Y        ;LOAD A INDIRECT INDEXED
```

2. Store accumulator indexed, assuming that we have saved A at the top of the stack.

```
        LDY     INDEX           ;GET LSB OF INDEX
        PLA                     ;RESTORE A
        STA     (TEMP),Y        ;STORE A INDIRECT INDEXED
```

· **Autopreincrementing**. Autopreincrementing means that the contents of the index register are incremented automatically before they are used. You can provide autopreincrementing on the 6502 processor either by using INX or INY on an index register or by using the 16-bit methods to increment a base address in memory.

Examples

· Load the accumulator from address BASE using autopreincrementing on index register X.

```
        INX                     ;AUTOPREINCREMENT X
        LDA     BASE,X
```

We assume that the array contains fewer than 256 elements.

· Load the accumulator from the address in memory locations PGZRO and PGZRO + 1 using autopreincrementing on the contents of memory locations INDEX and INDEX + 1.

```
        INC     INDEX           ;AUTOPREINCREMENT INDEX
        BNE     DONE
        INC     INDEX+1         ;WITH CARRY IF NECESSARY
DONE    LDA     PGZRO           ;MOVE LSB
        STA     TEMP
        LDA     PGZRO+1         ;ADD MSB'S
        CLC
        ADC     INDEX+1
        STA     TEMP+1
        LDY     INDEX           ;GET LSB OF INDEX
        LDA     (TEMP),Y        ;LOAD ACCUMULATOR
```

If you must autoincrement by 2 (as in handling arrays of addresses) use the sequence

```
        LDA     INDEX           ;AUTOINCREMENT INDEX BY 2
        CLC
        ADC     #2
        STA     INDEX
        BCC     DONE
        INC     INDEX+1         ;CARRY TO MSB IF NECESSARY
DONE    NOP
```

· **Autopostincrementing**. Autopostincrementing means that the contents of the index register are incremented automatically after they are used. You can provide autopreincrementing on the 6502 processor either by using INX or INY on an index register or by using the 16-bit methods to increment an index in memory.

Examples

· Load the accumulator from address BASE using autopostincrementing on index register Y.

```
        LDA     BASE,Y          ;AUTOPOSTINCREMENT Y
        INY
```

· Load the accumulator from the address in memory locations PGZRO and PGZRO + 1 using autopostincrementing on the contents of memory locations INDEX and INDEX + 1.

```
        LDA     PGZRO           ;MOVE LSB OF BASE
        STA     TEMP
        LDA     PGZRO+1         ;ADD MSB'S OF BASE AND INDEX
        CLC
        ADC     INDEX+1
        STA     TEMP+1
        LDY     INDEX           ;GET LSB OF INDEX
        LDA     (TEMP),Y        ;LOAD ACCUMULATOR
        INC     INDEX           ;AUTOPOSTINCREMENT INDEX
        BNE     DONE
        INC     INDEX+1         ;WITH CARRY IF NECESSARY
DONE    NOP
```

· **Autopredecrementing**. Autopredecrementing means that the contents of the index register are decremented automatically before they are used. You can provide autopredecrementing on the 6502 processor either by using DEX or DEY on an index register or by using the 16-bit methods to decrement a base address or index in memory.

Examples

· Load the accumulator from address BASE using autopredecrementing on index register X.

```
        DEX                     ;AUTOPREDECREMENT X
        LDA     BASE,X
```

We assume that the array contains fewer than 256 elements.

· Load the accumulator from the address in memory locations PGZRO and PGZRO + 1 using autopredecrementing on the contents of memory locations INDEX and INDEX + 1.

```
        LDA     INDEX       ;AUTOPREDECREMENT INDEX
        BNE     DECLSB
        DEC     INDEX+1     ;BORROWING FROM MSB IF NECESSARY
DECLSB  DEC     INDEX
        LDA     PGZRO       ;MOVE LSB OF BASE
        STA     TEMP
        LDA     PGZRO+1     ;ADD MSB'S OF BASE AND INDEX
        CLC
        ADC     INDEX+1
        STA     TEMP+1
        LDY     INDEX       ;GET LSB OF INDEX
        LDA     (TEMP),Y    ;LOAD ACCUMULATOR
```

If you must autodecrement by 2 (as in handling arrays of addresses), use the sequence:

```
        LDA     INDEX       ;AUTODECREMENT INDEX BY 2
        SEC
        SBC     #2
        STA     INDEX
        BCS     DONE
        DEC     INDEX+1     ;BORROWING FROM MSB IF NECESSARY
DONE    NOP
```

· **Autopostdecrementing**. Autopostdecrementing means that the contents of the index register are decremented automatically after they are used. You can provide autopostdecrementing on the 6502 processor by using either DEX or DEY on an index register or by using the 16-bit methods to decrement an index in memory.

Examples

· Load the accumulator from address BASE using autopostdecrementing on index register Y.

```
        LDA     BASE,Y      ;AUTOPOSTDECREMENT Y
        DEY
```

· Load the accumulator from the address in memory locations PGZRO and PGZRO + 1 using autopostdecrementing on the contents of memory locations INDEX and INDEX + 1.

```
        LDA     PGZRO       ;MOVE LSB OF BASE
        STA     TEMP
        LDA     PGZRO+1     ;ADD MSB'S OF BASE AND INDEX
        CLC
        ADC     INDEX+1
        STA     TEMP+1
        LDY     INDEX       ;GET LSB OF INDEX
        LDA     (TEMP),Y    ;LOAD ACCUMULATOR
        CPY     #0          ;AUTOPOSTDECREMENT INDEX
        BNE     DECLSB
        DEC     INDEX+1     ;BORROWING FROM MSB IF NECESSARY
DECLSB  DEC     INDEX
```

· **Indexed indirect addressing (preindexing)**. The 6502 processor provides preindexing for many instructions. We can simulate preindexing for the instructions that lack it by moving the data to the accumulator using preindexing, operating on it, and (if necessary) storing the result back into memory using preindexing.

Examples

1. Rotate right the data at the preindexed address obtained by indexing with X from base address PGZRO.

```
LDA     (PGZRO,X)    ;GET THE DATA
ROR     A            ;ROTATE DATA RIGHT
STA     (PGZRO,X)    ;STORE RESULT BACK IN MEMORY
```

2. Clear the preindexed address obtained by indexing with X from base address PGZRO.

```
LDA     #0           ;DATA = ZERO
STA     (PGZRO,X)    ;CLEAR PREINDEXED ADDRESS
```

Note that if the calculation of an effective address in preindexing produces a result too large for eight bits, the excess is truncated and no error warning occurs. That is, the processor provides an automatic wraparound on page 0.

· **Indirect indexed addressing (postindexing)**. The 6502 processor provides postindexing for many instructions. We can simulate postindexing for the instructions that lack it by moving the data to the accumulator using postindexing, operating on it, and (if necessary) storing the result back into memory using postindexing.

Examples

1. Decrement the data at the address in memory locations PGZRO and PGZRO+1 using Y as an index.

```
LDA     (PGZRO),Y    ;GET THE DATA
SEC
SBC     #1           ;DECREMENT DATA BY 1
STA     (PGZRO),Y    ;STORE RESULT BACK IN MEMORY
```

2. Rotate left the data at the address in memory locations PGZRO and PGZRO+1 using Y as an index.

```
LDA     (PGZRO),Y    ;GET THE DATA
ROL     A            ;ROTATE DATA LEFT
STA     (PGZRO),Y    ;STORE RESULT BACK IN MEMORY
```

REFERENCES

1. Osborne, A. *An Introduction to Microcomputers, Volume 1: Basic Concepts,* 2nd ed. Berkeley: Osborne/McGraw-Hill, 1980.

2. Leventhal, L.A. *6800 Assembly Language Programming.* Berkeley: Osborne/McGraw-Hill, 1978.

3. Leventhal, L.A. *6809 Assembly Language Programming.* Berkeley: Osborne/McGraw-Hill, 1981.

4. Fischer, W.P. "Microprocessor Assembly Language Draft Standard," *IEEE Computer,* December 1979, pp. 96-109.

5. Scanlon, L.J. *6502 Software Design,* Howard W. Sams, Indianapolis, Ind., 1980, pp. 111-13.

Chapter 3 Common Programming Errors

This chapter describes common errors in 6502 assembly language programs. The final section describes common errors in input/output drivers and interrupt service routines. Our aims here are the following:

- To warn programmers of potential trouble spots and sources of confusion.
- To indicate likely causes of programming errors.
- To emphasize some of the techniques and warnings presented in Chapters 1 and 2.
- To inform maintenance programmers where to look for errors and misinterpretations.
- To provide the beginner with a starting point in the difficult process of locating and correcting errors.

Of course, no list of errors can be complete. We have emphasized the most common ones in our work, but we have not attempted to describe the rare, subtle, or occasional errors that frustrate even the experienced programmer. However, most errors are remarkably simple once you uncover them and this list should help you debug most programs.

CATEGORIZATION OF PROGRAMMING ERRORS

We may generally divide common 6502 programming errors into the following categories:

- Using the Carry improperly. Typical errors include forgetting to clear the Carry before addition or set it before subtraction, and interpreting it incorrectly after comparisons (it acts as an inverted borrow).

· Using the other flags improperly. Typical errors include using the wrong flag (such as Negative instead of Carry), branching after instructions that do not affect a particular flag, inverting the branch conditions (particularly when the Zero flag is involved), and changing a flag accidentally before branching.

· Confusing addresses and data. Typical errors include using immediate instead of direct addressing, or vice versa, and confusing memory locations on page 0 with the addresses accessed indirectly through those locations.

· Using the wrong formats. Typical errors include using BCD (decimal) instead of binary, or vice versa, and using binary or hexadecimal instead of ASCII.

· Handling arrays incorrectly. Typical problems include accidentally overrunning the array at one end or the other (often by 1) and ignoring page boundaries when the array exceeds 256 bytes in length.

· Ignoring implicit effects. Typical errors include using the contents of the accumulator, index register, stack pointer, flags, or page 0 locations without considering the effects of intermediate instructions on these contents. Most errors arise from instructions that have unexpected, implicit, or indirect effects.

· Failing to provide proper initial conditions for routines or for the microcomputer as a whole. Most routines require the initialization of counters, indirect addresses, indexes, registers, flags, and temporary storage locations. The microcomputer as a whole requires the initialization of the Interrupt Disable and Decimal Mode flags and all global RAM addresses (note particularly indirect addresses and other temporary storage on page 0).

· Organizing the program incorrectly. Typical errors include skipping or repeating initialization routines, failing to update indexes, counters, or indirect addresses, and forgetting to save intermediate or final results.

A common source of errors, one that is beyond the scope of our discussion, is conflict between user programs and systems programs. A simple example is a user program that saves results in temporary storage locations that operating systems or utility programs need for their own purposes. The results thus disappear mysteriously even though a detailed trace of the user program does not reveal any errors.

More complex sources of conflict may include the interrupt system, input/output ports, the stack, or the flags. After all, the systems programs must employ the same resources as the user programs. (Systems programs generally attempt to save and restore the user's environment, but they often have subtle or unexpected effects.) Making an operating system transparent to the user is a problem comparable to devising a set of regulations, laws, or tax codes that have no loopholes or side effects.

USING THE CARRY IMPROPERLY

The following instructions and conventions are the most common sources of errors:

· CMP, CPX, and CPY affect the Carry as if it were an inverted borrow, that is, they set the Carry if the subtraction of the memory location from the register did not require a borrow, and they clear the Carry if it did. Thus, Carry = 1 if no borrow was necessary and Carry = 0 if a borrow was required. This is contrary to the sense of the Carry in most other microprocessors (the 6800, 6809, 8080, 8085, or Z-80).

· SBC subtracts the inverted Carry flag from the normal subtraction of the memory location from the accumulator. That is, it produces the result (A) − (M) − (1 − Carry). If you do not want the Carry flag to affect the result, you must set it with SEC. Like comparisons, SBC affects the Carry as if it were an inverted borrow; Carry = 0 if the subtraction requires a borrow and 1 if it does not.

· ADC always includes the Carry in the addition. This produces the result (A) = (A) + (M) + Carry. If you do not want the Carry flag to affect the result, you must clear it with CLC. Note that the Carry has its normal meaning after ADC.

Examples

1. CMP ADDR

This instruction sets the flags as if the contents of memory location ADDR had been subtracted from the accumulator. The Carry flag is set if the subtraction does not require a borrow and cleared if it does. Thus

Carry = 1 if (A) ≥ (ADDR)
Carry = 0 if (A) < (ADDR)

We are assuming that both numbers are unsigned. Note that the Carry is set (to 1) if the numbers are equal.

2. SBC #VALUE

This instruction subtracts VALUE and 1−Carry from the accumulator. It sets the flags just like a comparison. To subtract VALUE alone from the accumulator, you must use the sequence

```
SEC                 ;SET INVERTED BORROW
SBC     #VALUE      ;SUBTRACT VALUE
```

This sequence produces the result (A) = (A) − VALUE. If VALUE = 1, the sequence is equivalent to a Decrement Accumulator instruction (remember, DEC cannot be applied to A).

3. ADC #VALUE

This instruction adds VALUE and Carry to the accumulator. To add VALUE alone to the accumulator, you must use the sequence

```
CLC                 ;CLEAR CARRY
ADC     #VALUE      ;ADD VALUE
```

This sequence produces the result (A) = (A) + VALUE. If VALUE = 1, the sequence is equivalent to an Increment Accumulator instruction (remember, INC cannot be applied to A).

USING THE OTHER FLAGS INCORRECTLY

Instructions for the 6502 generally have expected effects on the flags. The only special case is BIT. Situations that require some care include the following:

- Store instructions (STA, STX, and STY) do not affect the flags, so the flags do not necessarily reflect the value that was just stored. You may need to test the register by transferring it to another register or comparing it with 0. Note that load instructions (including PHA) and transfer instructions (excluding TXS) affect the Zero and Negative flags.

- After a comparison (CMP, CPX, or CPY), the Zero flag indicates whether the operands are equal. The Zero flag is set if the operands are equal and cleared if they are not. There is some potential confusion here — BEQ means *branch if the result is equal to 0;* that is, *branch if the Zero flag is 1.* Be careful of the difference between the result being 0 and the Zero flag being 0. These two conditions are opposites; the Zero flag is 0 if the result is not 0.

- In comparing unsigned numbers, the Carry flag indicates which number is larger. CMP, CPX, or CPY clears the Carry if the register's contents are greater than or equal to the other operand and sets the Carry if the register's contents are less. Note that comparing equal operands sets the Carry. If these alternatives *(greater than or equal* and *less than)* are not what you need (you want the alternatives to be *greater than* and *less than or equal)*, you can reverse the subtraction, subtract 1 from the accumulator, or add 1 to the other operand.

- In comparing signed numbers, the Negative flag indicates which operand is larger unless two's complement overflow has occurred. We must first look at the Overflow flag. If that flag is 0, the Negative flag indicates which operand is larger; if that flag is 1, the sense of the Negative flag is inverted.

After a comparison (if no overflow occurs), the Negative flag is set if the register's contents are less than the other operand, and cleared if the register's

contents are greater than or equal to the other operand. Note that comparing equal operands clears the Negative flag. As with the Carry, you can handle the equality case in the opposite way by adjusting either operand or by reversing the subtraction.

· If a condition holds and you wish the computer to do something, a common procedure is to branch around a section of the program on the opposite condition. For example, to increment memory location OVFLW if the Carry is 1, use the sequence

```
        BCC     NEXT
        INC     OVFLW
NEXT    NOP
```

The branch condition is the opposite of the condition under which the section should be executed.

· Increment and decrement instructions do not affect the Carry flag. This allows the instructions to be used for counting in loops that perform multiple-byte arithmetic (the Carry is needed to transfer carries or borrows between bytes). Increment and decrement instructions do, however, affect the Zero and Negative flags; you can use the effect on the Zero flag to determine whether an increment has produced a carry. Note the following typical sequences:

1. 16-bit increment of memory locations INDEX and INDEX+1 (MSB in INDEX+1)

```
        INC     INDEX       ;INCREMENT LSB
        BNE     DONE
        INC     INDEX+1     ;AND CARRY TO MSB IF NECESSARY
DONE    NOP
```

We determine if a carry has been generated by examining the Zero flag after incrementing the less significant byte.

2. 16-bit decrement of memory locations INDEX and INDEX+1 (MSB in INDEX+1)

```
        LDA     INDEX       ;CHECK LSB
        BNE     DECLSB
        DEC     INDEX+1     ;BORROW FROM MSB IF NECESSARY
DECLSB  DEC     INDEX       ;DECREMENT MSB
```

We determine if a borrow will be generated by examining the less significant byte before decrementing it.

· The BIT instruction has rather unusual effects on the flags. It places bit 6 of the memory location in the Overflow flag and bit 7 in the Negative flag, regardless of the value in the accumulator. Thus, only the Zero flag actually reflects the logical ANDing of the accumulator and the memory location.

· Only a few instructions affect the Carry or Overflow flags. The instructions that affect Carry are arithmetic (ADC, SBC), comparisons (CMP, CPX, and CPY), and shifts (ASL, LSR, ROL, and ROR), besides the obvious CLC and SEC. The only instructions that affect Overflow are ADC, BIT, CLV, and SBC; comparison and shift instructions do not affect the Overflow flag, unlike the situation in the closely related 6800 and 6809 microprocessors.

Examples

1. The sequence

```
        STA     $1700
        BEQ     DONE
```

will have unpredictable results, since STA does not affect any flags. Sequences that will produce a jump if the value stored is 0 are

```
        STA     $1700
        CMP     #0              ;TEST ACCUMULATOR
        BEQ     DONE
```

or

```
        STA     $1700
        TAX                     ;TEST ACCUMULATOR
        BEQ     DONE
```

2. The instruction CMP #$25 sets the Zero flag as follows:

Zero = 1 if the contents of A are 25_{16}
Zero = 0 if the contents of A are not 25_{16}

Thus, if you want to increment memory location COUNT, if (A) = 25_{16}, use the sequence

```
        CMP     #$25            ;IS A 25?
        BNE     DONE
        INC     COUNT           ;YES, INCREMENT COUNT
DONE    NOP
```

Note that we use BNE to branch around the increment if the condition (A = 25_{16}) does not hold. It is obviously easy to err by inverting the branch condition.

3. The instruction CPX #$25 sets the Carry flag as follows:

Carry = 0 if the contents of X are between 00 and 24_{16}
Carry = 1 if the contents of X are between 25_{16} and FF_{16}

Thus, the Carry flag is cleared if X contains an unsigned number less than the other operand and set if X contains an unsigned number greater than or equal to the other operand.

If you want to clear the Carry if the X register contains 25_{16}, use CPX #$26 instead of CPX #$25. That is, we have

```
CPX     #$25
BCC     LESS          ;BRANCH IF (X) LESS THAN 25
```

or

```
CPX     #$26
BCC     LESSEQ        ;BRANCH IF (X) 25 OR LESS
```

4. The sequence SEC, SBC #$40 sets the Negative (Sign) flag as follows:

Negative = 0 if A is between 40_{16} and $7F_{16}$ (normal signed arithmetic) or if A is between 80_{16} and $C0_{16}$ (because of two's complement overflow)

Negative = 1 if A is between 00_{16} and $3F_{16}$ or between $C1_{16}$ and FF_{16} (normal signed arithmetic)

Two's complement overflow occurs if A contains a number between 80_{16} (-128_{10} in two's complement) and $C0_{16}$ (-64_{10} in two's complement). Then subtracting 40_{16} (64_{10}) produces a result less than -128_{10}, which is beyond the range of an 8-bit signed number. The setting of the Overflow flag indicates this out-of-range condition.

The following sequence will thus produce a branch if A contains a signed number less than 40_{16}.

```
SEC                   ;SET INVERTED BORROW
SBC     #$40          ;SUBTRACT 40 HEX
BVS     DEST          ;BRANCH IF OVERFLOW IS SET
BMI     DEST          ;OR IF DIFFERENCE IS NEGATIVE
```

Note that we cannot use CMP here, since it does not affect the Overflow flag. We could, however, use the sequence

```
CMP     #0            ;BRANCH IF A IS NEGATIVE
BMI     DEST
CMP     #$40          ;OR IF A IS POSITIVE BUT BELOW 40 HEX
BCC     DEST
```

We eliminate the possibility of overflow by handling negative numbers separately.

5. The sequence

```
INC     ADDR
BCS     NXTPG
```

will have unpredictable results, since INC does not affect the Carry flag. A sequence that will produce a jump, if the result of the increment is 00 (thus implying the production of a carry), is illustrated below.

```
INC     ADDR
BEQ     NXTPG
```

We can tell when an increment has produced a carry, but we cannot tell when a decrement has required a borrow since the result then is FF_{16}, not 0. Thus, it is much simpler to increment a multibyte number than to decrement it.

6. The sequence

```
BIT     ADDR
BVS     DEST
```

produces a branch if bit 6 of ADDR is 1. The contents of the accumulator do not affect it. Similarly, the sequence

```
BIT     ADDR
BPL     DEST
```

produces a branch if bit 7 of ADDR is 0. The contents of the accumulator do not affect it. The only common sequence with BIT in which the accumulator matters is

```
LDA     #MASK
BIT     ADDR
```

This sequence sets the Zero flag if logically ANDing MASK and the contents of ADDR produces a result of 0. A typical example using the Zero flag is

```
LDA     #%00010000
BIT     ADDR
BNE     DEST          ;BRANCH IF BIT 4 OF ADDR IS 1
```

This sequence forces a branch if the result of the logical AND is nonzero, that is, if bit 4 of ADDR is 1.

The effects of BIT on the Overflow and Negative flags do not generally cause programming errors since there are no standard, widely used effects that might cause confusion. These effects do, however, create documentation problems since the approach is unique and those unfamiliar with the 6502 cannot be expected to guess what is happening.

7. The sequence

```
CMP     #VALUE
BVS     DEST
```

produces unpredictable results, since CMP does not affect the Overflow flag. Instead, to produce a branch if the subtraction results in two's complement overflow, use the sequence

```
SEC                   ;SET INVERTED BORROW
SBC     #VALUE        ;SUBTRACT VALUE
BVS     DEST          ;BRANCH IF OVERFLOW OCCURS
```

CONFUSING ADDRESSES AND DATA

The rules to remember are

· The immediate addressing mode requires the actual data as an operand. That is, LDA #$40 loads the accumulator with the number 40_{16}.

· The absolute and zero page (direct) addressing modes require the address of the data as an operand. That is, LDA $40 loads the accumulator with the contents of memory location 0040_{16}.

· The indirect indexed and indexed indirect addressing modes obtain the indirect address from two memory locations on page 0. The indirect address is in two memory locations starting at the specified address; it is stored *upside-down,* with its less significant byte at the lower address. Fortunately, the indexed indirect (preindexed) mode is rarely used and is seldom a cause of errors. The meaning of addressing modes with JMP and JSR can be confusing, since these instructions use addresses as if they were data. The assumption is that one could not transfer control to a number, so a jump with immediate addressing would be meaningless. However, the instruction JMP $1C80 loads $1C80_{16}$ into the program counter, just like a load with immediate addressing, even though we conventionally say that the instruction uses absolute addressing. Similarly, the instruction JMP (ADDR) loads the program counter with the address from memory locations ADDR and ADDR+1; it thus acts like a load instruction with absolute (direct) addressing.

Examples

1. LDX#$20 loads the number 20_{16} into index register X. LDX $20 loads the contents of memory location 0020_{16} into index register X.

2. LDA ($40),Y loads the accumulator from the address obtained by indexing with Y from the base address in memory locations 0040_{16} and 0041_{16} (MSB in 0041_{16}). Note that if LDA ($40),Y makes sense, then LDA ($41),Y generally does not, since it uses the base address in memory locations 0041_{16} and 0042_{16}. Thus, the indirect addressing modes generally make sense only if the indirect addresses are aligned properly on word boundaries; however, the 6502 does not check this alignment in the way that many computers (particularly IBM machines) do. The programmer must make sure that all memory locations used indirectly contain addresses with the bytes arranged properly.

Confusing addresses and their contents is a frequent problem in handling data structures. For example, the queue of tasks to be executed by a piece of test equipment might consist of a block of information for each task. That block might contain

· The starting address of the test routine.

- The number of seconds for which the test is to run.
- The address in which the result is to be saved.
- The upper and lower thresholds against which the result is to be compared.
- The address of the next block in the queue.

Thus, the block contains data, direct addresses, and indirect addresses. Typical errors that a programmer could make are

- Transferring control to the memory locations containing the starting address of the test routine, rather than to the actual starting address.
- Storing the result in the block rather than in the address specified in the block.
- Using a threshold as an address rather than as data.
- Assuming that the next block starts within the current block, rather than at the address given in the current block.

Jump tables are another common source of errors. The following are alternative implementations:

- Form a table of jump instructions and transfer control to the correct element (for example, to the third jump instruction).
- Form a table of destination addresses and transfer control to the contents of the correct element (for example, to the address in the third element).

You will surely have problems if you try to use the jump instructions as indirect addresses or if you try to execute the indirect addresses.

FORMAT ERRORS

The rules you should remember are

- A $ in front of a number (or an H at the end) indicates hexadecimal to the assembler and a % in front or a B at the end indicates binary. Be careful — some assemblers use different symbols.
- The default mode of most assemblers is decimal; that is, most assemblers assume all numbers to be decimal unless they are specifically designated as something else. A few assemblers (such as Apple's miniassembler and the mnemonic entry mode in Rockwell's AIM-65) assume hexadecimal as a default.
- ADC and SBC instructions produce decimal results if the Decimal Mode flag is 1 and binary results if the Decimal Mode flag is 0. All other instructions, including DEC, DEX, DEY, INC, INX, and INY, always produce binary results.

You should make special efforts to avoid the following common errors:

- Omitting the hexadecimal designation ($ or H) from a hexadecimal data item or address. The assembler will assume the item to be a decimal number if it contains no letter digits. It will treat the item as a name if it is valid (it must start with a letter in most assemblers). The assembler will indicate an error only if the item cannot be interpreted as a decimal number or a name.

- Omitting the binary designation (% or B) from a binary data item. The assembler will assume it to be a decimal number.

- Confusing decimal (BCD) representations with binary representations. Remember, ten is not an integral power of two, so the binary and BCD representations are not the same beyond nine. Standard BCD constants must be designated as hexadecimal numbers, not as decimal numbers.

- Confusing binary or decimal representations with ASCII representations. An ASCII input device produces ASCII characters and an ASCII output device responds to ASCII characters.

Examples

1. LDA 2000

This instruction loads the accumulator from memory address 2000_{10} ($07D0_{16}$), not address 2000_{16}. The assembler will not produce an error message, since 2000 is a valid decimal number.

2. AND #00000011

This instruction logically ANDs the accumulator with the decimal number 11 (1011_2), not with the binary number 11 (3_{10}). The assembler will not produce an error message, since 00000011 is a valid decimal number despite its unusual form.

3. ADC #40

This instruction adds 40_{10} (not $40_{16} = 64_{10}$) and the Carry to the accumulator. Note that 40_{10} is not the same as 40 BCD, which is 40_{16}; $40_{10} = 28_{16}$. The assembler will not produce an error message, since 40 is a valid decimal number.

4. LDA #3

This instruction loads the accumulator with the number 3. If this value is now sent to an ASCII output device, it will respond as if it had received the character ETX (03_{16}), not the character 3 (33_{16}). The correct version is

```
LDA     #'3          ;GET AN ASCII 3
```

5. If memory location 0040_{16} contains a single digit, the sequence

```
LDA     $40
STA     PORT
```

will not print that digit on an ASCII output device. The correct sequence is

```
LDA     $40           ;GET DECIMAL DIGIT
CLC
ADC     #'0           ;ADJUST TO ASCII
STA     PORT
```

or

```
LDA     $40           ;GET DECIMAL DIGIT
ORA     #%00110000    ;ADJUST TO ASCII
STA     PORT
```

6. If input port IPORT contains a single ASCII decimal digit, the sequence

```
LDA     IPORT
STA     $40
```

will not store the actual digit in memory location 0040_{16}. Instead, it will store the ASCII version, which is the actual digit plus 30_{16}. The correct sequence is

```
LDA     IPORT         ;GET ASCII DIGIT
SEC
SBC     #'0           ;ADJUST TO DECIMAL
STA     $40
```

or

```
LDA     IPORT         ;GET ASCII DIGIT
AND     #%11001111    ;ADJUST TO DECIMAL
STA     $40
```

Handling decimal arithmetic on the 6502 microprocessor is simple, since the processor has a Decimal Mode (D) flag. When that flag is set (by SED), all additions and subtractions produce decimal results. So, the following sequences implement decimal addition and subtraction:

- Decimal addition of memory location ADDR to the accumulator

```
SED                   :ENTER DECIMAL MODE
CLC
ADC     ADDR          ;ADD DECIMAL
CLD                   :LEAVE DECIMAL MODE
```

- Decimal subtraction of memory location ADDR from the accumulator

```
SED                   :ENTER DECIMAL MODE
SEC
SBC     ADDR          ;SUBTRACT DECIMAL
CLD                   ;LEAVE DECIMAL MODE
```

Since increment and decrement instructions always produce binary results, we must use the following sequences (assuming the D flag is set).

Increment memory location 0040_{16} in the decimal mode

```
LDA     $40
CLC
ADC     #1
STA     $40
```

Decrement memory location 0040_{16} in the decimal mode

```
LDA     $40
SEC
SBC     #1
STA     $40
```

The problem with the decimal mode is that it has implicit effects. That is, the same ADC and SBC instructions with the same data will produce different results, depending on the state of the Decimal Mode flag. The following procedures will reduce the likelihood of the implicit effects causing unforeseen errors:

- Initialize the Decimal Mode flag (with CLD) as part of the regular system initialization. Note that RESET has no effect on the Decimal Mode flag.
- Clear the Decimal Mode flag as soon as you are through performing decimal arithmetic.
- Initialize the Decimal Mode flag in interrupt service routines that include ADC or SBC instructions. That is, such service routines should execute CLD before performing any binary addition or subtraction.

HANDLING ARRAYS INCORRECTLY

The following situations are the most common sources of errors:

- If you are counting an index register down to 0, the zero index value may never be used. The solution is to reduce the base address or addresses by 1. For example, if the terminating sequence in a loop is

```
        DEX
        BNE LOOP
```

the processor will fall through as soon as X is decremented to 0. A typical adjusted loop (clearing NTIMES bytes of memory) is

```
        LDX     #NTIMES
        LDA     #0
CLEAR   STA     BASE-1,X
        DEX
        BNE     CLEAR
```

Note the use of BASE−1 in the indexed store instruction. The program clears addresses BASE through BASE + NTIMES−1.

· Although working backward through an array is often more efficient than working forward, programmers generally find it confusing. Remember that the address BASE+(X) contains the previous entry in a loop like the example shown above. Although the processor can work backward just as easily as it can work forward, programmers usually find themselves conditioned to thinking ahead.

· Be careful not to execute one extra iteration or stop one short. Remember, memory locations BASE through BASE+N contain N+1 entries, not N entries. It is easy to forget the last entry or, as shown above, drop the first one. On the other hand, if you have N entries, they will occupy memory locations BASE through BASE+N-1; now it is easy to find yourself working off the end of the array.

· You cannot extend absolute indexed addressing or zero-page indexed addressing beyond 256 bytes. If an index register contains FF_{16}, incrementing it will produce a result of 00. Similarly, if an index register contains 00, decrementing it will produce a result of FF_{16}. Thus, you must be careful about incrementing or decrementing index registers when you might accidentally exceed the capacity of eight bits. To extend loops beyond 256 bytes, use the indirect indexed (postindexed) addressing mode. Then the following sequence will add 1 to the more significant byte of the indirect address when index register Y is incremented to 0.

```
          INY                 ;INCREMENT INDEX REGISTER
          BNE    DONE
          INC    INDIR+1
DONE      NOP
```

Here INDIR and INDIR+1 are the locations on page 0 that contain the indirect address.

Example

1. Let us assume (INDIR) = 80_{16} and (INDIR+1) = $4C_{16}$, so that the initial base address is $4C80_{16}$. If the loop refers to the address (INDIR), Y, the effective address is (INDIR+1) (INDIR) + Y or $4C80_{16}$ + (Y). When Y = FF_{16}, the effective address is

$$4C80_{16} + (Y) = 4C80_{16} + FF_{16} = 4D7F_{16}$$

The sequence shown above for incrementing the index and the indirect address produces the results

(Y) = (Y) + 1 = 00
(INDIR+1) = (INDIR+1) = I = $4D_{16}$

The effective address for the next iteration will be

$$4D80_{16} + (Y) = 4D80_{16} = 00_{16} = 4D80_{16}$$

which is the next higher address in the normal consecutive sequence.

IMPLICIT EFFECTS

Some of the implicit effects you should remember are

- The changing of the Negative and Zero flags by load and transfer instructions, such as LDA, LDX, LDY, PLA, TAX, TAY, TSX, TXA, and TYA.
- The dependence of the results of ADC and SBC instructions on the values of the Carry and Decimal Mode flags.
- The special use of the Negative and Overflow flags by the BIT instruction.

The use of the memory address one larger than the specified one in the indirect, indirect indexed, and indexed indirect addressing modes.

- The changing of the stack pointer by PHA, PHP, PLA, PLP, JSR, RTS, RTI, and BRK. Note that JSR and RTS change the stack pointer by 2, and BRK and RTI change it by 3.
- The saving of the return address minus 1 by JSR and the addition of 1 to the restored address by RTS.
- The inclusion of the Carry in the rotate instructions ROL and ROR. The rotation involves nine bits, not eight bits.

Examples

1. LDX $40

This instruction affects the Negative and Zero flags, so those flags will no longer reflect the value in the accumulator or the result of the most recent operation.

2. ADC #$20

This instruction adds in the Carry flag as well as the immediate data (20_{16}). The result will be binary if the Decimal Mode flag is cleared, but BCD if the Decimal Mode flag is set.

3. BIT $1700

This instruction sets the Overflow flag from the value of bit 6 of memory location 1700_{16}. This is the only instruction that has a completely unexpected effect on that flag.

4. JMP ($1C00)

This instruction transfers control to the address in memory locations $1C00_{16}$ and $1C01_{16}$ (MSB in $1C01_{16}$). Note that $1C01_{16}$ is involved even though it is not specified, since indirect addresses always occupy two bytes of memory.

5. PHA

This instruction not only saves the accumulator in memory, but it also decrements the stack pointer by 1.

6. RTS

This instruction not only loads the program counter from the top two locations in the stack, but it also increments the stack pointer by 2 and the program counter by 1.

7. ROR A

This instruction rotates the accumulator right 1 bit, moving the former contents of bit position 0 into the Carry and the former contents of the Carry into bit position 7.

INITIALIZATION ERRORS

The initialization routines must perform the following tasks, either for the microcomputer system as a whole or for particular routines:

- Load all RAM locations with initial values. This includes indirect addresses and other temporary storage on page 0. You cannot assume that a memory location contains 0 just because you have not used it.
- Load all registers and flags with initial values. Reset initializes only the Interrupt Disable flag (to 1). Note, in particular, the need to initialize the Decimal Mode flag (usually with CLD) and the stack pointer (using the LDX, TXS sequence).
- Load all counters and indirect addresses with initial values. Be particularly careful of addresses on page 0 that are used in either the indirect indexed (postindexed) addressing mode or the indexed indirect (preindexed) mode.

ORGANIZING THE PROGRAM INCORRECTLY

The following problems are the most common:

- Failing to initialize a register, flag, or memory location. You cannot assume

that a register, flag, or memory location contains zero just because you have not used it.

· Accidentally reinitializing a register, flag, memory location, index, counter, or indirect address. Be sure that your branches do not cause some or all of the initialization instructions to be repeated.

· Failing to update indexes, counters, or indirect addresses. A problem here may be one path that branches around the updating instructions or changes some of the conditions before executing those instructions.

· Forgetting to save intermediate or final results. It is remarkably easy to calculate a result and then load something else into the accumulator. Errors like this are particularly difficult to locate, since all the instructions that calculate the result work properly and yet the result itself is being lost. A common problem here is for a branch to transfer control to an instruction that writes over the result that was just calculated.

· Forgetting to branch around instructions that should not be executed in a particular path. Remember, the computer will execute instructions consecutively unless told specifically to do otherwise. Thus, it is easy for a program to accidentally fall through to a section that the programmer expects it to reach only via a branch. An awkward feature of the 6502 is its lack of an unconditional relative branch; you must either use JMP with absolute addressing or set a condition and branch on it holding (SEC, BCS, DEST and CLV, BVC DEST).

ERROR RECOGNITION BY ASSEMBLERS

Most assemblers will immediately recognize the following common errors:

· Undefined operation code (usually a misspelling or an omission)
· Undefined name (often a misspelling or an omitted definition)
· Illegal character (for example, a 2 in a binary number or a B in a decimal number)
· Illegal format (for example, an incorrect delimiter or the wrong register or operand)
· Illegal value (usually a number too large for 8 or 16 bits)
· Missing operand
· Double definition (two different values assigned to one name)
· Illegal label (for example, a label attached to a pseudo-operation that does not allow a label)
· Missing label (for example, on an = pseudo-operation that requires one).

These errors are generally easy to correct. Often the only problem is an error, such as omitting the semicolon or other delimiter in front of a comment, that confuses the assembler and results in a series of meaningless error messages.

There are, however, many common errors that assemblers will not recognize. The programmer should be aware that his or her program may contain such errors even if the assembler does not report them. Typical examples are

· Omitted lines. Obviously, the assembler cannot identify a completely omitted line unless that line contains a label or definition that is used later in the program. The easiest lines to omit are repetitions (that is, one or more lines that are the same or sequences that start the same) or instructions that seem to be unnecessary. Typical repetitions are series of shifts, branches, increments, or decrements. Instructions that may appear unnecessary include CLC, SEC, and so forth.

· Omitted designations. The assembler cannot tell if you omitted a designation such as #, H, $, B, or % unless the omission results in an illegal character (such as C in a decimal number). Otherwise, the assembler will assume all addresses to be direct and all numbers to be decimal. Problems occur with numbers that are valid as either decimal or hexadecimal values (such as 44 or 2050) and with binary numbers (such as 00000110).

· Misspellings that are still valid. Typical examples are typing BCC instead of BCS, LDX instead of LDY, and SEC instead of SED. Unless the misspelling is invalid, the assembler has no way of knowing what you meant. Valid misspellings are often a problem if you use similar names or labels such as XXX and XXXX, L121 and L112, or VAR1I and VARII.

· Designating instructions as comments. If you place a semicolon at the start of an instruction line, the assembler will treat the line as a comment. This can be a perplexing error, since the line appears in the listing but is not assembled into object code.

Sometimes you can confuse the assembler by entering invalid instructions. An assembler may accept a totally illogical entry simply because its developer never considered such possibilities. The result can be unpredictable, much like the results of giving someone a completely wrong number (for example, a telephone number instead of a street address or a driver license number instead of a credit card number). Some cases in which a 6502 assembler can go wrong are

· If you designate an impossible register or addressing mode. Some assemblers will accept instructions like INC A, LDA ($40),X, or LDY BASE,Y. They will produce erroneous object code without any warning.

· If you enter an invalid digit, such as Q in a decimal or hexadecimal number or 7 in a binary number. Some assemblers will assign values to such erroneous digits in an aribitrary manner.

· If you enter an invalid operand such as LDA #$HX. Some assemblers will accept this and generate incorrect code.

The assembler will recognize only errors that its developer anticipated. Programmers are often able to make mistakes that the developer never imagined, much as automobile drivers are often capable of performing maneuvers that never occurred in the wildest dreams of a highway designer or traffic planner. Note that only a line-by-line hand checking of the program will find errors that the assembler does not recognize.

IMPLEMENTATION ERRORS

Occasionally, a microprocessor's instructions simply do not work the way the designers or anyone else would expect. The 6502 has one implementation error that is, fortunately, quite rare. The instruction JMP ($XXFF) where the Xs represent any page number, does not work correctly. One would expect this instruction to obtain the destination address from memory locations XXFF and (XX+1)00. Instead, it apparently does not increment the more significant byte of the indirect address; it therefore obtains the destination address from memory locations XXFF and XX00. For example, JMP ($1CFF) will jump to the address stored in memory locations $1CFF_{16}$ (LSB) and $1C00_{16}$ (MSB), surely a curious outcome. Most assemblers expect the programmer to ensure that no indirect jumps ever obtain their destination addresses across page boundaries.

COMMON ERRORS IN I/O DRIVERS

Most errors in I/O drivers involve both hardware and software, so they are often difficult to categorize. Some mistakes you should watch for are

· Confusing input ports and output ports. Many I/O interfaces use the READ/WRITE line for addressing, so that reading and writing the same memory address results in operations on different physical registers. Even when this is not done, it may still be impossible to read back output data unless it is latched and buffered.

· Attempting to perform operations that are physically impossible. Reading data from an output device (such as a display) or sending data to an input device (such as a keyboard) makes no physical sense. However, accidentally using the wrong address will cause no assembly errors; the address, after all, is valid and the assembler has no way of knowing that certain operations cannot be performed on it. Similarly, a program may attempt to save data in a nonexistent address or in a ROM.

· Forgetting implicit hardware effects. Sometimes transferring data to or from a port will change the status lines automatically, particularly if you are using a 6520 or 6522 parallel interface. Even reading or writing a port while debugging a program will change the status lines. Be particularly careful of instructions like comparisons and BIT which read a memory address even though they do not change any registers, and instructions like decrement, increment, and shift which both read and write a memory address (the actual operation, of course, takes place inside the processor).

· Reading or writing without checking status. Many devices can accept or provide data only when a status line indicates they are ready. Transferring data to or from them at other times will have unpredictable effects.

· Ignoring the differences between input and output. Remember that an input device normally starts out in the *not ready* state — it has no data available although the computer is ready to accept data. On the other hand, an output device normally starts out in the *ready* state, that is, it could accept data but the computer usually has none to send it. In many situations, particularly when using 6520, 6522, 6551, or 6850 devices, you may have to disable the outputs initially or send a null character (something that has no effect) to each output port just to change its state from *ready* to *not ready* initially.

· Failing to keep copies of output data. Remember that you may not be able to read the data back from the output port. If you need to repeat it later as part of repeating a transmission that was incorrectly received, change part of it (turn on or off one of several indicator lights attached to the same port), or save it as part of the interrupted status (the data is the current priority level). You must save a copy in memory. The copy must be updated every time the actual data is changed.

· Reading data before it is stable or while it is changing. Be sure that you understand exactly when the input device is guaranteed to produce stable data. In the case of switches that may bounce, you may want to sample them twice (more than a debouncing time apart) before taking any action. In the case of keys that may bounce, you may want to take action only when they are released rather than when they are pressed. The action on release also forces the operator to release the key rather than holding it down. In the case of persistent data (such as in serial I/O), you should center the reception, that is, read the data near the centers of the pulses rather than at the edges where the values may be changing.

· Forgetting to reverse the polarity of data being transferred to or from devices that operate in negative logic. Many simple I/O devices, such as switches and displays, use negative logic. A logic 0 means that a switch is closed or a display is lit. Common ten-position switches or dials also often produce data in negative logic, as do many encoders. The solution is simple — complement the data (using EOR #$FF) after reading it or before sending it.

· Confusing actual I/O ports with registers that are inside I/O devices. Programmable I/O devices, such as the 6520, 6522, 6551, and 6850, have control or command registers which determine how the device operates, and status registers that reflect the current state of the device or the transfer. These registers are inside the I/O devices; they are not connected to peripherals. Transferring data to or from status or control registers is not the same as transferring data to or from actual I/O ports.

· Using bidirectional ports improperly. Many devices, such as the 6520, 6522, 6530, and 6532, have bidirectional I/O ports. The ports (and perhaps even individual lines) can be used either as inputs or outputs. Normally, resetting the computer to avoid initial transients makes these ports inputs, so you must explicitly change them to outputs if necessary. Be cautious when reading bits or ports that are designated as outputs or writing into bits or ports that are designated as inputs. The only way to determine what will happen is to read the documentation for the specific device.

· Forgetting to clear status after performing an I/O operation. Once the processor has read data from an input port, that port should revert to the *not ready* state. Similarly, once the processor has written data into an output port, that port should revert to the *not ready* state. Some I/O devices change the status of their ports automatically after input or output operations, but others either do not or (as in the 6520) change status automatically only after input operations. Leaving the status set can result in an endless loop or highly erratic operation.

COMMON ERRORS IN INTERRUPT SERVICE ROUTINES

Many interrupt-related errors involve both hardware and software, but some of the common mistakes include the following:

· Failing to reenable interrupts during the service routine. The 6502 processor automatically disables interrupts after accepting one. It does reenable interrupts when RTI is executed, since RTI restores the status register from the stack.

· Failing to save and restore registers. The 6502 does not automatically save any registers except the program counter and the status register. So the accumulator, index registers, and scratchpad locations must be saved explicitly in the stack.

· Saving or restoring registers in the wrong order. Registers must be restored in the opposite order from that in which they were saved.

· Enabling interrupts before establishing priorities and other parameters of the interrupt system.

· Forgetting that the response to an interrupt includes saving the status register and the program counter at the top of the stack. The status register is on top and the program counter value is the actual return address, so the situation differs from subroutines in which the return address minus 1 is normally at the top of the stack.

· Not disabling the interrupt during multibyte transfers or instruction sequences that cannot be interrupted. In particular, you must avoid partial updating of data (such as time) that an interrupt service routine may use. In general, interrupts should be disabled when the main program is changing memory locations that it shares with interrupt service routines.

· Failing to reenable the interrupt after a sequence that must run with interrupts disabled. A corollary problem here is that you do not want to enable interrupts if they were not enabled when the sequence was entered. The solution is to save the previous state of the Interrupt Disable flag (using PHP) before executing the sequence and restore the previous state (using PLP) afterward. Note, however, that PLP restores the entire status register.

· Failing to initialize or establish the value of the Decimal Mode flag. An interrupt service routine should not assume a particular value (0) for the D flag. Instead, it should initialize that flag with CLD or SED if it executes ADC or SBC instructions. There is no need to save or restore the old D flag since that is done automatically as part of the saving and restoring of the status register. Initializing the D flag avoids problems if the service routine is entered from a program that runs with the D flag set.

· Failing to clear the signal that caused the interrupt. The service routine must clear the interrupt even if it does not require an immediate response or any input or output operations. Even when the processor has, for example, no data to send to an interrupting output device, it must still either clear the interrrupt or disable it. Otherwise, the processor will get caught in an endless loop. Similarly, a real-time clock interrupt will typically require no servicing other than an updating of time, but the service routine still must clear the clock interrupt. This clearing may involve reading a 6520 or 6522 I/O port or timer.

· Failing to communicate with the main program. The main program will not realize that the interrupt has been serviced unless it is informed explicitly. The usual way to inform the main program is to have the interrupt service routine change a flag that the main program can examine. The main program will then know that the service routine has been executed. The procedure is comparable to the practice of a postal patron raising a flag to indicate that he or she has mail to be picked up. The postman lowers the flag after picking up the mail. Note that this

simple procedure means that the main program must examine the flag often enough to avoid missing data or messages. Of course, the programmer can always provide an intermediate storage area (or buffer) that can hold many data items.

· Failing to save and restore priority. The priority of an interrupt is often held in a write-only register or in a memory location. That priority must be saved just like the registers and restored properly at the end of the service routine. If the priority register is write-only, a copy of its contents must be saved in memory.

Introduction to the Program Section

The program section contains sets of assembly language subroutines for the 6502 microprocessor. Each subroutine is documented with an introductory section and comments; each is followed by at least one example of its use. The introductory material contains the following information:

1. Purpose of the routine
2. Procedure followed
3. Registers used
4. Execution time
5. Program size
6. Data memory required
7. Special cases
8. Entry conditions
9. Exit conditions
10. Examples

We have made each routine as general as possible. This is most difficult in the case of the input/output (I/O) and interrupt service routines described in Chapters 10 and 11, since in practice these routines are always computer-dependent. In such cases, we have limited the computer dependence to generalized input and output handlers and interrupt managers. We have drawn specific examples there from the popular Apple II computer, but the general principles are applicable to other 6502-based computers as well.

In all routines, we have used the following parameter passing techniques:

1. A single 8-bit parameter is passed in the accumulator. A second 8-bit parameter is passed in index register Y.

2. A single 16-bit parameter is passed in the accumulator and index register Y with the more significant byte in the accumulator. An accompanying 8-bit parameter is passed in index register X.

3. Larger numbers of parameters are passed in the stack, either directly or indirectly. We assume that subroutines are entered via a JSR instruction that places the return address at the top of the stack, and hence on top of the parameters.

Where there has been a choice between execution time and memory usage, we have chosen the approach that minimizes execution time. For example, in the case of arrays that are more than 256 bytes long, it is faster to handle the full pages, then handle the remaining partial page separately, than to handle the entire array in a single loop. The reason is that the first approach can use an 8-bit counter in an index register, whereas the second approach requires a 16-bit counter in memory.

We have also chosen the approach that minimizes the number of repetitive calculations. For example, in the case of array indexing, the number of bytes between the starting addresses of elements differing only by one in a particular subscript (known as the *size* of that subscript) depends only on the number of bytes per element and the bounds of the array. Thus, the sizes of the various subscripts can be calculated as soon as the bounds of the array are known; the sizes are therefore used as parameters for the indexing routines, so that they need not be calculated each time a particular array is indexed.

As for execution time, we have specified it for most short routines. For longer routines, we have given an approximate execution time. The execution time of programs involving many branches will obviously depend on which path is followed in a particular case. This is further complicated for the 6502 by the fact that branch instructions themselves require different numbers of clock cycles depending on whether the branch is not taken, taken within the current page, or taken across a page boundary. Thus, a precise execution time is often impossible to define. The documentation always contains at least one typical example showing an approximate or maximum execution time.

Our philosophy on error indications and special cases has been the following:

1. Routines should provide an easily tested indicator (such as the Carry flag) of whether any errors or exceptions have occurred.

2. Trivial cases, such as no elements in an array or strings of zero length, should result in immediate exits with minimal effect on the underlying data.

3. Misspecified data (such as a maximum string length of zero or an index beyond the end of an array) should result in immediate exits with minimal effect on the underlying data.

4. The documentation should include a summary of errors and exceptions (under the heading of "Special Cases").

5. Exceptions that may actually be convenient for the user (such as deleting more characters than could possibly be left in a string rather than counting the precise number) should be handled in a reasonable way, but should still be indicated as errors.

Obviously, no method of handling errors or exceptions can ever be completely consistent or well suited to all applications. We have taken the approach that a reasonable set of subroutines must deal with this issue, rather than ignoring it or assuming that the user will always provide data in the proper form.

The subroutines are listed as follows:

Code Conversion

Array Manipulation and Indexing

Arithmetic

Bit Manipulation and Shifts

String Manipulation

Array Operations

Input/Output

Interrupts

Binary to BCD Conversion (BN2BCD) 4A

Converts one byte of binary data to two bytes of BCD data.

Procedure: The program subtracts 100 repeatedly from the original data to determine the hundreds digit, then subtracts ten repeatedly from the remainder to determine the tens digit, and finally shifts the tens digit left four positions and combines it with the ones digit.

Registers Used: All

Execution Time: 133 cycles maximum, depends on the number of subtractions required to determine the tens and hundreds digits.

Program Size: 38 bytes

Data Memory Required: One byte anywhere in RAM (address TEMP).

Entry Conditions

Binary data in the accumulator.

Exit Conditions

Hundreds digit in the accumulator
Tens and ones digits in index register Y.

Examples

1. Data: (A) = $6E_{16}$ (110 decimal)

   Result: (A) = 01_{16} (hundreds digit)
   (Y) = 10_{16} (tens and ones digits)

2. Data: (A) = $B7_{16}$ (183 decimal)

   Result: (A) = 01_{16} (hundreds digit)
   (Y) = 83_{16} (tens and ones digits)

```
;
;
;
;
;       Title           Binary to BCD conversion
;       Name:           BN2BCD
;
;
;
;       Purpose:        Convert one byte of binary data to two
;                       bytes of BCD data
;
;       Entry:          Register A = binary data
;
;       Exit:           Register A = high byte of BCD data
;                       Register Y = low  byte of BCD data
;
```

```
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           133 cycles maximum                              ;
;                                                                       ;
;       Size:           Program 38 bytes                                ;
;                       Data     1 byte                                 ;
;                                                                       ;
;                                                                       ;

BN2BCD:
        ; CALCULATE 100'S DIGIT
        ;   DIVIDE BY 100
        ; Y = QUOTIENT
        ; A = REMAINDER
        LDY     #0FFH           ;START QUOTIENT AT -1
        SEC                     ;SET CARRY FOR INITIAL SUBTRACTION
D100LP:
        INY                     ;ADD 1 TO QUOTIENT
        SBC     #100            ;SUBTRACT 100
        BCS     D100LP          ;BRANCH IF A IS STILL LARGER THAN 100
        ADC     #100            ;ADD THE LAST 100 BACK
        TAX                     ;SAVE REMAINDER
        TYA
        PHA                     ;SAVE 100'S DIGIT ON THE STACK
        TXA                     ;GET REMAINDER

        ; CALCULATE 10'S AND 1'S DIGITS
        ;  DIVIDE REMAINDER OF THE 100'S DIGIT BY 10
        ; Y = 10'S DIGIT
        ; A = 1'S DIGIT
        LDY     #0FFH           ;START QUOTIENT AT -1
        SEC                     ;SET CARRY FOR INITIAL SUBTRACTION
D10LP:
        INY                     ;ADD 1 TO QUOTIENT
        SBC     #10
        BCS     D10LP           ;BRANCH IF A IS STILL LARGER THAN 10
        ADC     #10             ;ADD THE LAST 10 BACK

        ;COMBINE 1'S AND 10'S DIGITS
        STA     TEMP            ;SAVE 1'S DIGIT
        TYA                     ;GET 10'S DIGIT
        ASL     A
        ASL     A
        ASL     A
        ASL     A               ;MOVE 10'S TO HIGH NIBBLE OF A
        ORA     TEMP            ;OR IN THE 1'S DIGIT

        ;RETURN WITH Y = LOW BYTE A = HIGH BYTE
        TAY                     ;PLACE IN REG Y
        PLA                     ;GET 100'S DIGIT
        RTS

;DATA
TEMP:   .BLOCK  1               ;TEMPORARY USED TO COMBINE 1'S AND 10'S DIGITS
```

```
;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0401:
        ;CONVERT 0A HEXADECIMAL TO 10 BCD
        LDA     #0AH
        JSR     BN2BCD
        BRK                         ;A=0, Y=10H

        ;CONVERT FF HEXADECIMAL TO 255 BCD
        LDA     #0FFH
        JSR     BN2BCD
        BRK                         ;A=02H, Y=55H

        ;CONVERT 0 HEXADECIMAL TO 0 BCD
        LDA     #0
        JSR     BN2BCD
        BRK                         ;A=0, Y=0

        .END
```

BCD to Binary Conversion (BCD2BN) 4B

Converts one byte of BCD data to one byte of binary data.

Procedure: The program masks off the more significant digit, multiplies it by ten using shifts (10 = 8 + 2, and multiplying by eight or by two is equivalent to three or one left shifts, respectively), and adds the product to the less significant digit.

Registers Used: A, P, Y

Execution Time: 38 cycles

Program Size: 24 bytes

Data Memory Required: One byte anywhere in RAM (Address TEMP).

Entry Conditions

BCD data in the accumulator.

Exit Conditions

Binary data in the accumulator.

Examples

1. Data: (A) = 99_{16}

   Result: (A) = $63_{16} = 99_{10}$

2. Data: (A) = 23_{16}

   Result: (A) = $17_{16} = 23_{10}$

```
;
;
;
;       Title           BCD to binary conversion
;       Name:           BCD2BN
;
;
;
;       Purpose:        Convert one byte of BCD data to one
;                       byte of binary data
;
;       Entry:          Register A = BCD data
;
;       Exit:           Register A = Binary data
;
;       Registers used: A,P,Y
;
;       Time:           38 cycles
```

```
;                                                                       ;
;       Size:           Program 24 bytes                                ;
;                       Data     1  byte                                ;
;                                                                       ;
;                                                                       ;

BCD2BN:
        ;MULTIPLY UPPER NIBBLE BY 10 AND SAVE IT
        ; TEMP := UPPER NIBBLE * 10 WHICH EQUALS UPPER NIBBLE * (8 + 2)
        TAY                     ;SAVE ORIGINAL VALUE
        AND     #0F0H           ;GET UPPER NIBBLE
        LSR     A               ;DIVIDE BY 2 WHICH = UPPER NIBBLE * 8
        STA     TEMP            ;SAVE * 8
        LSR     A               ;DIVIDE BY 4
        LSR     A               ;DIVIDE BY 8: A = UPPER NIBBLE * 2
        CLC
        ADC     TEMP
        STA     TEMP            ;REG A = UPPER NIBBLE * 10

        TYA                     ;GET ORIGINAL VALUE
        AND     #0FH            ;GET LOWER NIBBLE
        CLC
        ADC     TEMP            ;ADD TO UPPER NIBBLE

        RTS

;DATA
TEMP:   .BLOCK  1


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0402:

        ;CONVERT 0 BCD TO 0 HEXADECIMAL
        LDA     #0
        JSR     BCD2BN
        BRK                     ;A=0

        ;CONVERT 99 BCD TO 63 HEXADECIMAL
        LDA     #099H
        JSR     BCD2BN
        BRK                     ;A=63H

        ;CONVERT 23 BCD TO 17 HEXADECIMAL
        LDA     #23H
        JSR     BCD2BN
        BRK                     ;A=17H

        .END
```

Binary to Hexadecimal ASCII Conversion (BN2HEX)

4C

Converts one byte of binary data to two ASCII characters corresponding to the two hexadecimal digits.

Procedure: The program masks off each hexadecimal digit separately and converts it to its ASCII equivalent. This involves a simple addition of 30_{16} if the digit is decimal. If the digit is non-decimal, an additional factor of seven must be added to handle the break between ASCII 9 (39_{16}) and ASCII A (41_{16}).

Registers Used: All

Execution Time: 77 cycles plus three extra cycles for each non-decimal digit.

Program Size: 31 bytes

Data Memory Required: None

Entry Conditions

Binary data in the accumulator.

Exit Conditions

ASCII equivalent of more significant hexadecimal digit in the accumulator
ASCII equivalent of less significant hexadecimal digit in index register Y.

Examples

1. Data: (A) = FB_{16}

   Result: (A) = 46_{16} (ASCII F)
   (Y) = 42_{16} (ASCII B)

2. Data: (A) = 59_{16}

   Result: (A) = 35_{16} (ASCII 5)
   (Y) = 39_{16} (ASCII 9)

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Binary to hex ASCII                          ;
;       Name:           BN2HEX                                       ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Purpose:        Convert one byte of binary data to           ;
;                       two ASCII characters                         ;
;                                                                    ;
;       Entry:          Register A = Binary data                     ;
;                                                                    ;
;       Exit:           Register A = First ASCII digit, high order value;
;                       Register Y = Second ASCII digit, low order value;
```

```
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Approximately 77 cycles                         ;
;                                                                       ;
;       Size:           Program 31 bytes                                ;
;                                                                       ;
;                                                                       ;

BN2HEX:
        ;
        ;CONVERT HIGH NIBBLE
        TAX                     ;SAVE ORIGINAL VALUE
        AND     #0F0H           ;GET HIGH NIBBLE
        LSR     A
        LSR     A
        LSR     A
        LSR     A               ;MOVE TO LOWER NIBBLE
        JSR     NASCII          ;CONVERT TO ASCII
        PHA                     ;SAVE IT ON THE STACK

        ;CONVERT LOW NIBBLE
        TXA
        AND     #0FH            ;GET LOW NIBBLE
        JSR     NASCII          ;CONVERT TO ASCII

        TAY                     ;LOW NIBBLE TO REG Y
        PLA                     ;HIGH NIBBLE TO REG A
        RTS

;
;SUBROUTINE NASCII
;PURPOSE:   CONVERT A HEXADECIMAL DIGIT TO ASCII
;ENTRY: A = BINARY DATA IN LOWER NIBBLE
;EXIT:  A = ASCII CHARACTER
;REGISTERS USED: A,P

;

NASCII:
        CMP     #10
        BCC     NAS1            ;BRANCH IF HIGH NIBBLE < 10
        CLC
        ADC     #7              ;ELSE ADD 7 SO AFTER ADDING '0' THE
                                ;  CHARACTER WILL BE IN 'A'..'F'
NAS1:
        ADC     #'0'            ;MAKE A CHARACTER
        RTS


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;
```

```
SC0403:
        ;CONVERT 0 TO '00'
        LDA     #0
        JSR     BN2HEX
        BRK                     ;A='0'=30H, Y='0'=30H

        ;CONVERT FF HEX TO 'FF'
        LDA     #0FFH
        JSR     BN2HEX
        BRK                     ;A='F'=46H, Y='F'=46H

        ;CONVERT 23 HEX TO '23'
        LDA     #23H
        JSR     BN2HEX
        BRK                     ;A='2'=32H, Y='3'=33H

        .END
```

Hexadecimal ASCII to Binary Conversion (HEX2BN)

4D

Converts two ASCII characters (representing two hexadecimal digits) to one byte of binary data.

Procedure: The program converts each ASCII character separately to a hexadecimal digit. This involves a simple subtraction of 30_{16} (ASCII zero) if the digit is decimal. If the digit is non-decimal, an additional factor of seven must be subtracted to handle the break between ASCII 9 (39_{16}) and ASCII A (41_{16}). The program then shifts the more significant digit left four bits and combines it with the less significant digit. The program does not check the validity of the ASCII characters (i.e., whether they are, in fact, the ASCII representations of hexadecimal digits).

Registers Used: A, P, Y

Execution Time: 74 cycles plus three extra cycles for each non-decimal digit.

Program Size: 30 bytes

Data Memory Required: One byte anywhere in RAM (address TEMP).

Entry Conditions

More significant ASCII digit in the accumulator, less significant ASCII digit in index register Y.

Exit Conditions

Binary data in the accumulator.

Examples:

1. Data: (A) = 44_{16} (ASCII D)
   (Y) = 37_{16} (ASCII 7)

   Result: (A) = $D7_{16}$

2. Data: (A) = 31_{16} (ASCII 1)
   (Y) = 42_{16} (ASCII B)

   Result: (A) = $1B_{16}$

```
;
;
;
;
;       Title           Hex ASCII to binary
;       Name:           HEX2BN
;
;
;
;       Purpose:        Convert two ASCII characters to one
;                       byte of binary data
;
```

```
;       Entry:          Register A = First ASCII digit, high order value;
;                       Register Y = Second ASCII digit, low order value;
;                                                                       ;
;       Exit:           Register A = Binary data                        ;
;                                                                       ;
;       Registers used: A,P,Y                                           ;
;                                                                       ;
;       Time:           Approximately 74 cycles                         ;
;                                                                       ;
;       Size:           Program 30 bytes                                ;
;                       Data     1 byte                                 ;
;                                                                       ;
;                                                                       ;

HEX2BN:
        PHA                     ;SAVE HIGH CHARACTER
        TYA                     ;GET LOW CHARACTER
        JSR     A2HEX           ;CONVERT IT
        STA     TEMP            ;SAVE LOW NIBBLE
        PLA                     ;GET THE HIGH CHARACTER
        JSR     A2HEX           ;CONVERT IT
        ASL     A
        ASL     A
        ASL     A
        ASL     A               ;SHIFT HIGH NIBBLE TO THE UPPER 4 BITS
        ORA     TEMP            ;OR IN THE LOW NIBBLE
        RTS

;
;SUBROUTINE: A2HEX
;PURPOSE: CONVERT ASCII TO A HEX NIBBLE
;ENTRY: A = ASCII CHARACTER
;EXIT:  A = BINARY VALUE OF THE ASCII CHARACTER
;REGISTERS USED: A,P
;

A2HEX:
        SEC                     ;SUBTRACT ASCII OFFSET
        SBC     #'0'
        CMP     #10
        BCC     A2HEX1          ;BRANCH IF A IS A DECIMAL DIGIT
        SBC     #7              ;ELSE SUBTRACT OFFSET FOR LETTERS
A2HEX1:
        RTS

;DATA
TEMP:   .BLOCK  1


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;
```

```
SC0404:
        ;CONVERT 'C7' TO C7 HEXADECIMAL
        LDA     #'C'
        LDY     #'7'
        JSR     HEX2BN
        BRK                         ;A=C7H

        ;CONVERT '2F' TO 2F HEXADECIMAL
        LDA     #'2'
        LDY     #'F'
        JSR     HEX2BN              ;A=2FH
        BRK

        ;CONVERT '23' TO 23 HEXADECIMAL
        LDA     #'2'
        LDY     #'3'
        JSR     HEX2BN
        BRK                         ;A=23H

        .END
```

Conversion of a Binary Number to Decimal ASCII (BN2DEC) 4E

Converts a 16-bit signed binary number to an ASCII string, consisting of the length of the number (in bytes), an ASCII minus sign (if necessary), and the ASCII digits.

Procedure: The program takes the absolute value of the number if it is negative and then keeps dividing by ten until it produces a quotient of zero. It converts each digit of the quotient to ASCII (by adding ASCII 0) and concatenates the digits along with an ASCII minus sign (in front) if the original number was negative.

Registers Used: All

Execution Time: Approximately 7,000 cycles

Program Size: 174 bytes

Data Memory Required: Seven bytes anywhere in RAM for the return address (two bytes starting at address RETADR), the sign of the original value (address NGFLAG), temporary storage for the original value (two bytes starting at address VALUE), and temporary storage for the value mod 10 (two bytes starting at address MOD10). Also, two bytes on page 0 for the buffer pointer (address BUFPTR, taken as $00D0_{16}$ and $00D1_{16}$ in the listing). This data memory does not include the output buffer which should be seven bytes long.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of output buffer address
More significant byte of output buffer address

Less significant byte of value to convert
More significant byte of value to convert

Exit Conditions

Order in buffer

Length of the string in bytes
ASCII − (if original number was negative)
ASCII digits (most significant digit first)

Examples

1. Data: Value to convert = $3EB7_{16}$

   Result (in output buffer):
   05 (number of bytes in buffer)
   31 (ASCII 1)
   36 (ASCII 6)
   30 (ASCII 0)
   35 (ASCII 5)
   35 (ASCII 5)

   That is, $3EB7_{16} = 16055_{10}$.

2. Data: Value to convert = $FFC8_{16}$

   Result (in output buffer):
   03 (number of bytes in buffer)
   2D (ASCII −)
   35 (ASCII 5)
   36 (ASCII 6)

   That is, $FFC8_{16} = -56_{10}$, when considered as a signed two's complement number.

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           Binary to decimal ASCII                         ;
;       Name:           BN2DEC                                          ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Convert a 16-bit signed binary number           ;
;                       to ASCII data                                   ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of the output buffer address,        ;
;                         High byte of the output buffer address,       ;
;                         Low byte of the value to convert,             ;
;                         High byte of the value to convert             ;
;                                                                       ;
;       Exit:           The first byte of the buffer is the length,     ;
;                       followed by the characters.                     ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Approximately 7,000 cycles                      ;
;                                                                       ;
;       Size:           Program 170 bytes                               ;
;                       Data      7 bytes plus                          ;
;                                 2 bytes in page zero                  ;
;                                                                       ;
;                                                                       ;

;PAGE ZERO POINTER
BUFPTR: .EQU    0D0H            ;PAGE ZERO BUFFER POINTER

;PROGRAM
BN2DEC:
        ;SAVE PARAMETERS
        PLA
        STA     RETADR          ;SAVE LOW BYTE OF RETURN ADDRESS
        PLA
        STA     RETADR+1        ;SAVE HIGH BYTE
        PLA
        STA     VALUE           ;SAVE LOW BYTE OF VALUE
        PLA
        STA     VALUE+1         ;SAVE HIGH BYTE OF THE VALUE TO CONVERT
        STA     NGFLAG          ;SAVE MSB OF VALUE AS SIGN OF VALUE
        BPL     GETBP           ;BRANCH IF VALUE IS POSITIVE
        LDA     #0              ;ELSE TAKE ABSOLUTE VALUE (0 - VALUE)
        SEC
        SBC     VALUE
        STA     VALUE
```

```
        LDA     #0
        SBC     VALUE+1
        STA     VALUE+1

GETBP:
        PLA                             ;SAVE STARTING ADDRESS OF OUTPUT BUFFER
        STA     BUFPTR
        PLA
        STA     BUFPTR+1


        ;SET BUFFER TO EMPTY
        LDA     #0
        LDY     #0                      ;BUFFER[0] := 0
        STA     (BUFPTR),Y

        ;
        ;CONVERT VALUE TO A STRING
CNVERT:
        ;VALUE := VALUE DIV 10
        ;MOD10 := VALUE MOD 10
        LDA     #0
        STA     MOD10
        STA     MOD10+1

        LDX     #16
        CLC                             ;CLEAR CARRY
DVLOOP:
        ROL     VALUE                   ;SHIFT THE CARRY INTO DIVIDEND BIT 0
        ROL     VALUE+1                 ;WHICH WILL BE THE QUOTIENT
        ROL     MOD10                   ;AND SHIFT DIVIDEND AT THE SAME TIME
        ROL     MOD10+1

        ;
        ;A,Y = DIVIDEND - DIVISOR
        SEC
        LDA     MOD10
        SBC     #10
        TAY                             ;SAVE LOW BYTE IN REG Y
        LDA     MOD10+1
        SBC     #0                      ;SUBTRACT CARRY
        BCC     DECCNT                  ;BRANCH IF DIVIDEND < DIVISOR
        STY     MOD10                   ;ELSE
        STA     MOD10+1                 ;  NEXT BIT OF QUOTIENT IS A ONE AND SET
                                        ;  DIVIDEND := DIVIDEND - DIVISOR

DECCNT:
        DEX
        BNE     DVLOOP

        ROL     VALUE                   ;SHIFT IN THE LAST CARRY FOR THE QUOTIENT
        ROL     VALUE+1

        ;CONCATENATE THE NEXT CHARACTER
```

```
CONCH:
        LDA     MOD10
        CLC
        ADC     #'0'            ;CONVERT 0..9 TO ASCII '0'..'9'
        JSR     CONCAT

        ;IF VALUE <> 0 THEN CONTINUE
        LDA     VALUE
        ORA     VALUE+1
        BNE     CNVERT          ;BRANCH IF VALUE IS NOT ZERO

EXIT:
        LDA     NGFLAG
        BPL     POS             ;BRANCH IF ORIGINAL VALUE WAS POSITIVE
        LDA     #'-'            ;ELSE
        JSR     CONCAT          ; PUT A MINUS SIGN IN FRONT

POS:
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA
        RTS                     ;RETURN

;
;SUBROUTINE: CONCAT
;PURPOSE: CONCATENATE THE CHARACTER IN REGISTER A TO THE
;         FRONT OF THE STRING ACCESSED THROUGH BUFPTR
;ENTRY: BUFPTR[0] = LENGTH
;EXIT:  REGISTER A CONCATENATED (PLACED IMMEDIATELY AFTER THE LENGTH BYTE)
;REGISTERS USED: A,P,Y
;

CONCAT:
        PHA                     ;SAVE THE CHARACTER ON THE STACK

        ;MOVE THE BUFFER RIGHT ONE CHARACTER
        LDY     #0
        LDA     (BUFPTR),Y      ;GET CURRENT LENGTH
        TAY
        BEQ     EXITMR          ;BRANCH IF LENGTH = 0
MVELP:
        LDA     (BUFPTR),Y      ;GET NEXT CHARACTER
        INY
        STA     (BUFPTR),Y      ;STORE IT
        DEY
        DEY
        BNE     MVELP           ;CONTINUE UNTIL DONE

EXITMR:
        PLA                     ;GET THE CHARACTER BACK FROM THE STACK
        LDY     #1
        STA     (BUFPTR),Y      ;STORE THE CHARACTER
        LDY     #0
        LDA     (BUFPTR),Y      ;GET LENGTH BYTE
```

```
        CLC
        ADC     #1              ;INCREMENT LENGTH BY 1
        STA     (BUFPTR),Y      ;UPDATE LENGTH

        RTS

;DATA
RETADR: .BLOCK  2               ;SAVE RETURN ADDRESS
NGFLAG: .BLOCK  1               ;SIGN OF ORIGINAL VALUE
VALUE:  .BLOCK  2               ;VALUE TO CONVERT
MOD10:  .BLOCK  2               ;MODULO 10 TEMPORARY


;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION:                                            ;
;                                                                    ;
;                                                                    ;


SC0405:
        ;CONVERT 0 TO '0'
        LDA     BUFADR+1        ;HIGH BYTE OF BUFFER ADDRESS
        PHA
        LDA     BUFADR          ;LOW BYTE BUFFER ADDRESS
        PHA
        LDA     VALUE1+1        ;HIGH BYTE OF VALUE
        PHA
        LDA     VALUE1          ;LOW BYTE OF VALUE
        PHA
        JSR     BN2DEC          ;CONVERT
        BRK                     ;BUFFER SHOULD = '0'

        ;CONVERT 32767 TO '32767'
        LDA     BUFADR+1        ;HIGH BYTE OF BUFFER ADDRESS
        PHA
        LDA     BUFADR          ;LOW BYTE BUFFER ADDRESS
        PHA
        LDA     VALUE2+1        ;HIGH BYTE OF VALUE
        PHA
        LDA     VALUE2          ;LOW BYTE OF VALUE
        PHA
        JSR     BN2DEC          ;CONVERT
        BRK                     ;BUFFER SHOULD = '32767'

        ;CONVERT -32768 TO '-32768'
        LDA     BUFADR+1        ;HIGH BYTE OF BUFFER ADDRESS
        PHA
        LDA     BUFADR          ;LOW BYTE BUFFER ADDRESS
        PHA
        LDA     VALUE3+1        ;HIGH BYTE OF VALUE
        PHA
        LDA     VALUE3          ;LOW BYTE OF VALUE
        PHA
```

```
        JSR     BN2DEC          ;CONVERT
        BRK                     ;BUFFER SHOULD = '-32768'
        JMP     SC0405

VALUE1: .WORD   0               ;TEST VALUE 1
VALUE2: .WORD   32767           ;TEST VALUE 2
VALUE3: .WORD   -32768          ;TEST VALUE 3
BUFADR: .WORD   BUFFER          ;BUFFER ADDRESS
BUFFER: .BLOCK  7               ;7 BYTE BUFFER

        .END
```

Conversion of ASCII Decimal to Binary (DEC2BN)

4F

Converts an ASCII string consisting of the length of the number (in bytes), a possible ASCII − or + sign, and a series of ASCII digits to two bytes of binary data. Note that the length is an ordinary binary number, not an ASCII number.

Procedure: The program sets a flag if the first ASCII character is a minus sign and skips over a leading plus sign. It then converts each subsequent digit to decimal (by subtracting ASCII zero), multiplies the previous digits by ten (using the fact that $10=8+2$, so a multiplication by ten can be reduced to left shifts and additions), and adds the new digit to the product. Finally, the program subtracts the result from zero if the original number was negative. The program exits immediately, setting the Carry flag, if it finds something other than a leading sign or a decimal digit in the string.

Registers Used: All

Execution Time: 670 cycles (approximately)

Program Size: 171 bytes

Data Memory Required: Four bytes anywhere in RAM for an index, a two-byte accumulator (starting address ACCUM), and a flag indicating the sign of the number (address NGLAG), two-bytes on page zero for a pointer to the string (address BUFPTR, taken as $00F0_{16}$ and $00F1_{16}$ in the listing).

Special Cases:

1. If the string contains something other than a leading sign or a decimal digit, the program returns with the Carry flag set to 1. The result in registers A and Y is invalid.

2. If the string contains only a leading sign (ASCII + or ASCII −), the program returns with the Carry flag set to 1 and a result of zero.

Entry Conditions

(A) = More significant byte of string address
(Y) = Less significant byte of string address

Exit Conditions

(A) = More significant byte of binary value
(Y) = Less significant byte of binary value

Carry flag is 0 if the string was valid; Carry flag is 1 if the string contained an invalid character. Note that the result is a signed two's complement 16-bit number.

Examples

1. Data: String consists of
   04 (number of bytes in string)
   31 (ASCII 1)
   32 (ASCII 2)
   33 (ASCII 3)
   34 (ASCII 4)
   That is, the number is $+1,234_{10}$.

   Result: (A) = 04_{16} (more significant byte of binary data)
   (Y) = $C2_{16}$ (less significant byte of binary data)
   That is, the number $+1234_{10} = 04C2_{16}$.

2. Data: String consists of
   06 (number of bytes in string)
   2D (ASCII −)
   33 (ASCII 3)
   32 (ASCII 2)
   37 (ASCII 7)
   35 (ASCII 5)
   30 (ASCII 0)
   That is, the number is $-32{,}750_{10}$.

   Result: (A) = 80_{16} (more significant byte of binary data)
   (Y) = 12_{16} (less significant byte of binary data)
   That is, the number $-32{,}750_{10} = 8012_{16}$.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Decimal ASCII to binary                      ;
;       Name:           DEC2BN                                       ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Purpose:        Convert ASCII characters to two bytes of binary ;
;                       data.                                        ;
;                                                                    ;
;       Entry:          Register A = high byte of string address     ;
;                       Register Y = low byte of string adddress     ;
;                       The first byte of the string is the length of ;
;                       the string.                                  ;
;                                                                    ;
;       Exit:           Register A = High byte of the value          ;
;                       Register Y = Low byte of the value           ;
;                       IF NO ERRORS THEN                            ;
;                         CARRY FLAG = 0                             ;
;                       ELSE                                         ;
;                         CARRY FLAG = 1                             ;
;                                                                    ;
;       Registers used: All                                          ;
;                                                                    ;
;       Time:           Approximately 670 cycles                     ;
;                                                                    ;
;       Size:           Program 171 bytes                            ;
;                       Data      4 bytes plus                       ;
;                                 2 bytes in page zero               ;
;                                                                    ;

;PAGE ZERO LOCATION
BUFPTR: .EQU    0F0H            ;PAGE ZERO POINTER TO STRING
;
;PROGRAM
DEC2BN:
        STA     BUFPTR+1
        STY     BUFPTR          ;SAVE THE STRING ADDRESS
```

```
        ;INITIALIZE
        LDY     #0
        LDA     (BUFPTR),Y   ;GET LENGTH
        TAX                  ; TO REGISTER X
        LDA     #1
        STA     INDEX        ;INDEX := 1
        LDA     #0
        STA     ACCUM        ;ACCUM := 0
        STA     ACCUM+1
        STA     NGFLAG       ;SIGN OF NUMBER IS POSITIVE

        ;CHECK THAT THE BUFFER IS NOT ZERO
        TXA
        BNE     INIT1        ;EXIT WITH ACCUM = 0 IF BUFFER IS EMPTY
        JMP     EREXIT       ;ERROR EXIT IF NOTHING IN BUFFER
INIT1:
        LDY     INDEX
        LDA     (BUFPTR),Y   ;GET FIRST CHARACTER
        CMP     #'-'         ;IS IT A MINUS ?
        BNE     PLUS         ;BRANCH IF NOT '-'
        LDA     #0FFH
        STA     NGFLAG       ;ELSE SIGN OF NUMBER IS NEGATIVE
        INC     INDEX        ;SKIP PAST MINUS SIGN
        DEX                  ;DECREMENT COUNT
        BEQ     EREXIT       ;ERROR EXIT IF ONLY '-' IN BUFFER
        JMP     CNVERT       ;START CONVERSION
PLUS:
        CMP     #'+'
        BNE     CHKDIG       ;START CONVERSION IF FIRST CHARACTER IS NOT '+'
        INC     INDEX
        DEX                  ;DECREMENT COUNT, IGNORE PLUS SIGN
        BEQ     EREXIT       ;ERROR EXIT IF ONLY '+' IN BUFFER

CNVERT:
        LDY     INDEX
        LDA     (BUFPTR),Y   ;GET NEXT CHARACTER
CHKDIG: CMP     #'0'
        BMI     EREXIT       ;ERROR IF < '0' (NOT A DIGIT)
        CMP     #'9'+1
        BPL     EREXIT       ;ERROR IF > '9' (NOT A DIGIT)
        PHA                  ;SAVE THE DIGIT ON THE STACK

        ;VALID DECIMAL DIGIT SO
        ;  ACCUM := ACCUM * 10
        ;    = ACCUM * (8 + 2)
        ;    = (ACCUM * 8) + (ACCUM * 2)
        ASL     ACCUM
        ROL     ACCUM+1      ;TIMES 2
        LDA     ACCUM
        LDY     ACCUM+1      ;SAVE ACCUM * 2
        ASL     ACCUM
        ROL     ACCUM+1
        ASL     ACCUM
        ROL     ACCUM+1      ;TIMES 8
        CLC
```

```
        ADC     ACCUM           ;SUM WITH * 2
        STA     ACCUM
        TYA
        ADC     ACCUM+1
        STA     ACCUM+1         ;ACCUM := ACCUM * 10

        ;ADD IN THE NEXT DIGIT
        ; ACCUM := ACCUM + DIGIT
        PLA                     ;GET THE DIGIT BACK
        SEC
        SBC     #'0'            ;CONVERT '0'..'9' TO BINARY 0..9
        CLC
        ADC     ACCUM
        STA     ACCUM
        BCC     D2B1            ;BRANCH IF NO CARRY TO HIGH BYTE
        INC     ACCUM+1         ;ELSE INCREMENT HIGH BYTE
D2B1:
        INC     INDEX           ;INCREMENT TO NEXT CHARACTER
        DEX
        BNE     CNVERT          ;CONTINUE CONVERSION

        LDA     NGFLAG
        BPL     OKEXIT          ;BRANCH IF THE VALUE WAS POSITIVE
        LDA     #0              ;ELSE REPLACE RESULT WITH -RESULT
        SEC
        SBC     ACCUM
        STA     ACCUM
        LDA     #0
        SBC     ACCUM+1
        STA     ACCUM+1

        ;GET THE BINARY VALUE AND RETURN
OKEXIT:
        CLC
        BCC     EXIT

EREXIT:
        SEC

EXIT:
        LDA     ACCUM+1         ;GET HIGH BYTE OF VALUE
        LDY     ACCUM
        RTS

;DATA
INDEX:  .BLOCK  1               ;INDEX INTO THE STRING
ACCUM:  .BLOCK  2               ;ACCUMULATED VALUE (2 BYTES)
NGFLAG: .BLOCK  1               ;SIGN OF NUMBER


;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;
```

```
SC0406:
        ;CONVERT '1234' TO 04D2 HEX
        LDA     ADRS1+1
        LDY     ADRS1           ;AY = ADDRESS OF S1
        JSR     DEC2BN
        BRK                     ;A = 04, Y = D2 HEX

        ;CONVERT '-32767' TO 7FFF HEX
        LDA     ADRS2+1
        LDY     ADRS2           ;AY = ADDRESS OF S2
        JSR     DEC2BN
        BRK                     ;A = 7F, Y = FF HEX

        ;CONVERT '-32768' TO 8000 HEX
        LDA     ADRS3+1
        LDY     ADRS3           ;AY = ADDRESS OF S3
        JSR     DEC2BN
        BRK                     ;A = 80 HEX, Y = 00 HEX

S1:     .BYTE   4,'1234'
S2:     .BYTE   6,'+32767'
S3:     .BYTE   6,'-32768'

ADRS1:  .WORD   S1              ;ADDRESS OF S1
ADRS2:  .WORD   S2              ;ADDRESS OF S2
ADRS3:  .WORD   S3              ;ADDRESS OF S3

        .END
```

Lower-Case to Upper-Case Translation (LC2UC) 4G

Converts an ASCII lower-case letter to its upper-case equivalent.

Procedure: The program determines from comparisons whether the data is an ASCII lower-case letter. If it is, the program subtracts 20_{16} from it, thus converting it to its upper-case equivalent. If it is not, the program leaves it unchanged.

Registers Used: A, P

Execution Time: 18 cycles if the original character is valid, fewer cycles otherwise.

Program Size: 12 bytes

Data Memory Required: None

Entry Conditions

Character in the accumulator.

Exit Conditions

If the character is an ASCII lower-case letter, the upper-case equivalent is in the accumulator. If the character is not an ASCII lower-case letter, the accumulator is unchanged.

Examples

1. Data: (A) = 62_{16} (ASCII b)
   Result: (A) = 42_{16} (ASCII B)

2. Data: (A) = 74_{16} (ASCII t)
   Result: (A) = 54_{16} (ASCII T)

```
;
;
;
;
;       Title           Lower case to upper case translation
;       Name:           LC2UC
;
;
;
;       Purpose:        Convert one ASCII character to upper case from
;                       lower case if necessary.
;
;       Entry:          Register A = Lower case ASCII character
;
```

```
;       Exit:           Register A = Upper case ASCII character if A    ;
;                                    is lower case, else A is unchanged.;
;                                                                       ;
;       Registers used: A,P                                             ;
;                                                                       ;
;       Time:           18 cycles if A is lower case, less otherwise    ;
;                                                                       ;
;       Size:           Program 12 bytes                                ;
;                       Data    none                                    ;
;                                                                       ;
;                                                                       ;

LC2UC:
        CMP     #'a'
        BCC     $1              ;BRANCH IF < 'a'
        CMP     #'z'+1
        BCS     EXIT            ;BRANCH IF > 'z'
        SEC
        SBC     #20H            ;CHANGE 'a'..'z' into 'A'..'Z'
EXIT:
        RTS

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0407:
        ;CONVERT LOWER CASE E TO UPPER CASE
        LDA     #'e'
        JSR     LC2UC
        BRK                     ;A='E'=45H

        ;CONVERT LOWER CASE Z TO UPPER CASE
        LDA     #'z'
        JSR     LC2UC
        BRK                     ;A='Z'=5AH

        ;CONVERT UPPER CASE A TO UPPER CASE A
        LDA     #'A'
        JSR     LC2UC
        BRK                     ;A='A'=41H


        .END    ;OF PROGRAM
```

ASCII to EBCDIC Conversion (ASC2EB) 4H

Converts an ASCII character to its EBCDIC equivalent.

Procedure: The program uses a simple table lookup with the data as the index and address EBCDIC as the base. Printable ASCII characters for which there are no EBCDIC equivalents are translated to an EBCDIC space (40_{16}); nonprintable ASCII characters without EBCDIC equivalents are translated to an EBCDIC NUL (00_{16}).

Registers Used: A, P, Y

Execution Time: 14 cycles

Program Size: Seven bytes, plus 128 bytes for the conversion table.

Data Memory Required: None

Entry Conditions

ASCII character in the accumulator.

Exit Conditions

EBCDIC equivalent in the accumulator.

Examples

1. Data: (A) = 35_{16} (ASCII 5)

   Result: (A) = $F5_{16}$ (EBCDIC 5)

2. Data: (A) = 77_{16} (ASCII w)

   Result: (A) = $A6_{16}$ (EBCDIC w)

3. Data: (A) = $2A_{16}$ (ASCII *)

   Result: (A) = $5C_{16}$ (EBCDIC *)

```
;
;
;
;
;       Title           ASCII to EBCDIC conversion
;       Name:           ASC2EB
;
;
;
;       Purpose:        Convert an ASCII character to its
;                       corresponding EBCDIC character
;
;       Entry:          Register A = ASCII character
;
;       Exit:           Register A = EBCDIC character
;
```

```
;       Registers used: A,P,Y                                                   ;
;                                                                               ;
;       Time:           14 cycles                                               ;
;                                                                               ;
;       Size:           Program 7 bytes                                         ;
;                       Data    128 bytes for the table                         ;
;                                                                               ;
;                                                                               ;


ASC2EB:
        AND     #7FH            ;BE SURE BIT 7 = 0
        TAY                     ;USE ASCII AS INDEX INTO EBCDIC TABLE
        LDA     EBCDIC,Y        ;GET EBCDIC
        RTS

;ASCII TO EBCDIC TABLE
; PRINTABLE ASCII CHARACTERS FOR WHICH THERE ARE NO EBCDIC EQUIVALENTS
; ARE TRANSLATED TO AN EBCDIC SPACE (040H), NON PRINTABLE ASCII CHARACTERS
; WITH NO EQUIVALENTS ARE TRANSLATED TO A EBCDIC NUL (000H)
EBCDIC:
;               NUL  SOH  STX  ETX  EOT  ENG  ACK  BEL          ;ASCII
        .BYTE   000H,000H,000H,022H,037H,000H,000H,000H          ;EBCDIC
;               BS   HT   LF   VT   FF   CR   SO   SI           ;ASCII
        .BYTE   000H,02BH,025H,000H,000H,02DH,000H,000H          ;EBCDIC
;               DLE  DC1  DC2  DC3  DC4  NAK  SYN  ETB          ;ASCII
        .BYTE   000H,000H,000H,000H,037H,000H,000H,000H          ;EBCDIC
;               CAN  EM   SUB  ESC  FS   GS   RS   VS           ;ASCII
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H          ;EBCDIC
;               SPACE !    "    #    $    %    &    '          ;ASCII
        .BYTE   040H,05AH,07EH,040H,05BH,06CH,050H,07CH          ;EBCDIC
;                (    )    *    +    ,    -    .    /          ;ASCII
        .BYTE   04DH,05DH,05CH,04EH,06BH,060H,04BH,061H          ;EBCDIC
;                0    1    2    3    4    5    6    7          ;ASCII
        .BYTE   0F0H,0F1H,0F2H,0F3H,0F4H,0F5H,0F6H,0F7H          ;EBCDIC
;                8    9    :    ;    <    =    >    ?          ;ASCII
        .BYTE   0F8H,0F9H,07AH,05EH,04CH,07DH,06EH,06FH          ;EBCDIC
;                @    A    B    C    D    E    F    G          ;ASCII
        .BYTE   07BH,0C1H,0C2H,0C3H,0C4H,0C5H,0C6H,0C7H          ;EBCDIC
;                H    I    J    K    L    M    N    O          ;ASCII
        .BYTE   0C8H,0C9H,0D1H,0D2H,0D3H,0D4H,0D5H,0D6H          ;EBCDIC
;                P    Q    R    S    T    U    V    W          ;ASCII
        .BYTE   0D7H,0D8H,0D9H,0E2H,0E3H,0E4H,0E5H,0E6H          ;EBCDIC
;                X    Y    Z    [    \    ]    ^    <-         ;ASCII
        .BYTE   0E7H,0E8H,0E9H,040H,040H,040H,06AH,040H          ;EBCDIC
;                `    a    b    c    d    e    f    g          ;ASCII
        .BYTE   07CH,081H,082H,083H,084H,085H,086H,087H          ;EBCDIC
;                h    i    j    k    l    m    n    o          ;ASCII
        .BYTE   088H,089H,091H,092H,093H,094H,095H,096H          ;EBCDIC
;                p    q    r    s    t    u    v    w          ;ASCII
        .BYTE   097H,098H,099H,0A2H,0A3H,0A4H,0A5H,0A6H          ;EBCDIC
;                x    y    z    {    |    }    ~    DEL        ;ASCII
        .BYTE   0A7H,0A8H,0A9H,040H,04FH,040H,05FH,007H          ;EBCDIC
```

```
;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0408:
        ;CONVERT ASCII 'A'
        LDA     #'A'            ;ASCII 'A'
        JSR     ASC2EB
        BRK                     ;EBCDIC 'A' = 0C1H

        ;CONVERT ASCII 'l'
        LDA     #'l'            ;ASCII 'l'
        JSR     ASC2EB
        BRK                     ;EBCDIC 'l' = 0F1H

        ;CONVERT ASCII 'a'
        LDA     #'a'            ;ASCII 'a'
        JSR     ASC2EB
        BRK                     ;EBCDIC 'a' = 081H

        .END            ;END PROGRAM
```

EBCDIC to ASCII Conversion (EB2ASC) 41

Converts an EBCDIC character to its ASCII equivalent.

Procedure: The program uses a simple table lookup with the data as the index and address ASCII as the base. Printable EBCDIC characters for which there are no ASCII equivalents are translated to an ASCII space (20_{16}); nonprintable EBCDIC characters without ASCII equivalents are translated to an ASCII NUL (00_{16}).

Registers Used: A, P, Y

Execution Time: 12 cycles

Program Size: Five bytes, plus 256 bytes for the conversion table.

Data Memory Required: None

Entry Conditions

EBCDIC character in the accumulator.

Exit Conditions

ASCII equivalent in the accumulator.

Examples

1. Data: (A) = 85_{16} (EBCDIC e)
   Result: (A) = 65_{16} (ASCII e)

2. Data: (A) = $4E_{16}$ (EBCDIC +)
   Result: (A) = $2B_{16}$ (ASCII +)

```
;
;
;
;
;       Title           EBCDIC to ASCII conversion
;       Name:           EB2ASC
;
;
;
;       Purpose:        Convert an EBCDIC character to its
;                       corresponding ASCII character
;
;       Entry:          Register A = EBCDIC character
;
;       Exit:           Register A = ASCII character
;
;       Registers used: A,P,Y
;
;       Time:           12 cycles
;
```

```
;       Size:           Program 5 bytes                                 ;
;                       Data    256 bytes for the table                 ;
;                                                                       ;
;                                                                       ;


EB2ASC:
        TAY
        LDA     ASCII,Y ;TRANSLATE
        RTS

;EBCDIC TO ASCII TABLE
; PRINTABLE EBCDIC CHARACTERS FOR WHICH THERE ARE NO ASCII EQUIVALENTS
; ARE TRANSLATED TO AN ASCII SPACE (020H), NON PRINTABLE EBCDIC CHARACTERS
; WITH NO EQUIVALENTS ARE TRANSLATED TO A ASCII NUL (000H)
ASCII:
;               NUL                             TAB         DEL         ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,009H,000H,07FH                 ;ASCII
;                                                                       ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H                 ;ASCII
;                                       NEW LINE                        ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,00DH,000H,000H                 ;ASCII
;                                                                       ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H                 ;ASCII
;                ETX                    LF                              ;EBCDIC
        .BYTE   000H,003H,000H,000H,000H,00AH,000H,000H                 ;ASCII
;                               TAB     CR                              ;EBCDIC
        .BYTE   000H,000H,000H,009H,000H,00DH,000H,000H                 ;ASCII
;                                                       EOT             ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,004H                 ;ASCII
;                                                                       ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H                 ;ASCII
;               SPACE                                                   ;EBCDIC
        .BYTE   ' ' ,000H,000H,000H,000H,000H,000H,000H                 ;ASCII
;                    BRITISH $  .    <    (    +    |                  ;EBCDIC
        .BYTE   000H,000H,' ' ,'.' ,'<' ,'(' ,'+' ,'|'                  ;ASCII
;                     &                                                 ;EBCDIC
        .BYTE   '&' ,000H,000H,000H,000H,000H,000H,000H                 ;ASCII
;                          !    $    *    )    ;    ~                   ;EBCDIC
        .BYTE   000H,000H,'!' ,'$' ,'*' ,')' ,';' ,'~'                  ;ASCII
;                         -    /                                        ;EBCDIC
        .BYTE   '-' ,'/' ,000H,000H,000H,000H,000H,000H                 ;ASCII
;                              ^         ,    %    -.-  >    ?          ;EBCDIC
        .BYTE   000H,000H,'^' ,',' ,'%' ,040H,'>' ,'?'                  ;ASCII
;                                                                       ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H                 ;ASCII
;                         :    @    '    =    "                        ;EBCDIC
        .BYTE   000H,000H,':' ,'@' ,''' ,'=' ,'"', 000H                 ;ASCII
;                     a    b    c    d    e    f    g                   ;EBCDIC
        .BYTE   000H,'a' ,'b' ,'c' ,'d' ,'e' ,'f' ,'g'                  ;ASCII
;                    h    i                                             ;EBCDIC
        .BYTE   'h' ,'i' ,000H,000H,000H,000H,000H,000H                 ;ASCII
;                         j    k    l    m    n    o    p               ;EBCDIC
        .BYTE   000H,'j' ,'k' ,'l' ,'m' ,'n' ,'o' ,'p'                  ;ASCII
;                    q    r                                             ;EBCDIC
```

```
        .BYTE   'q' ,'r' ,000H,000H,000H,000H,000H,000H         ;ASCII
;                        s    t    u    v    w    x           ;EBCDIC
        .BYTE   000H,000H,'s' ,'t' ,'u' ,'v' ,'w', 'x'          ;ASCII
;                y    z                                       ;EBCDIC
        .BYTE   'y' ,'z' ,000H,000H,000H,000H,000H,000H         ;ASCII
;                                                            ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H         ;ASCII
;                                                            ;EBCDIC
        .BYTE   000H,000H,000H,000H,000H,000H,000H,000H         ;ASCII
;                     A    B    C    D    E    F    G         ;EBCDIC
        .BYTE   000H,'A' ,'B' ,'C' ,'D' ,'E' ,'F' ,'G'          ;ASCII
;                H    I                                       ;EBCDIC
        .BYTE   'H' ,'I' ,000H,000H,000H,000H,000H,000H         ;ASCII
;                     J    K    L    M    N    O    P         ;EBCDIC
        .BYTE   000H,'J' ,'K' ,'L' ,'M' ,'N' ,'O' ,'P'          ;ASCII
;                Q    R                                       ;EBCDIC
        .BYTE   'Q' ,'R' ,000H,000H,000H,000H,000H,000H         ;ASCII
;                          S    T    U    V    W    X         ;EBCDIC
        .BYTE   000H,000H,'S' ,'T' ,'U' ,'V' ,'W', 'X'          ;ASCII
;                Y    Z                                       ;EBCDIC
        .BYTE   'Y' ,'Z' ,000H,000H,000H,000H,000H,000H         ;ASCII
;                0    1    2    3    4    5    6    7         ;EBCDIC
        .BYTE   '0' ,'1' ,'2' ,'3' ,'4' ,'5' ,'6' ,'7'          ;ASCII
;                9                                           ;EBCDIC
        .BYTE   '9' ,000H,000H,000H,000H,000H,000H,000H         ;ASCII

;                                                                   ;
;                                                                   ;
;       SAMPLE EXECUTION:                                           ;
;                                                                   ;
;                                                                   ;

SC0409:
        ;CONVERT EBCDIC 'A'
        LDA     #0C1H           ;EBCDIC 'A'
        JSR     EB2ASC
        BRK                     ;ASCII 'A' = 041H

        ;CONVERT EBCDIC '1'
        LDA     #0F1H           ;EBCDIC '1'
        JSR     EB2ASC
        BRK                     ;ASCII '1' = 031H

        ;CONVERT EBCDIC 'a'
        LDA     #081H           ;EBCDIC 'a'
        JSR     EB2ASC
        BRK                     ;ASCII 'a' = 061H

        .END            ;END PROGRAM
```

Memory Fill (MFILL) 5A

Places a specified value in each byte of a memory area of known size, starting at a given address.

Procedure: The program fills all the whole pages with the specified value first and then fills the remaining partial page. This approach is faster than dealing with the entire area in one loop, since 8-bit counters can be used instead of a 16-bit counter. The approach does, however, require somewhat more memory than a single loop with a 16-bit counter. A size of 0000_{16} causes an exit with no memory changed.

Registers Used: All

Execution Time: Approximately 11 cycles per byte plus 93 cycles overhead.

Program Size: 68 bytes

Data Memory Required: Five bytes anywhere in RAM for the array size (two bytes starting at address ARYSZ), the value (one byte at address VALUE), and the return address (two bytes starting at address RETADR). Also two bytes on page 0 for an array pointer (taken as addresses $00D0_{16}$ and $00D1_{16}$ in the listing).

Special Cases:

1. A size of zero causes an immediate exit with no memory changed.
2. Filling areas occupied or used by the program itself will cause unpredictable results. Obviously, filling any part of page 0 requires caution, since both this routine and most systems programs use that page.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Value to be placed in memory

Less significant byte of area size (in bytes)
More significant byte of area size (in bytes)

Less significant byte of starting address
More significant byte of starting address

Exit Conditions

The area from the starting address through the number of bytes given by the area size is filled with the specified value. The area filled thus starts at BASE and continues through BASE + SIZE − 1 (BASE is the starting address and SIZE is the area size).

Examples

1. Data: Value = FF_{16}
   Area size (in bytes) = 0380_{16}
   Starting address = $1AE0_{16}$

   Result: FF_{16} is placed in memory addresses $1AE0_{16}$ through $1E5F_{16}$.

2. Data: Value = EA_{16} (6502 operation code for NOP)
   Area size (in bytes) = $1C65_{16}$
   Starting address = $E34C_{16}$

   Result: EA_{16} is placed in memory addresses $E34C_{16}$ through $FFB0_{16}$

```
;       Title           Memory fill                                     ;
;       Name:           MFILL                                           ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:        Fill an area of memory with a value             ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Value to be placed in memory,                 ;
;                         Low byte of area size in bytes,               ;
;                         High byte of area size in bytes,              ;
;                         Low byte of starting address,                 ;
;                         High byte of starting address                 ;
;                                                                       ;
;       Exit:           Area filled with value                          ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Approximately 11 cycles per byte plus           ;
;                       93 cycles overhead.                             ;
;                                                                       ;
;       Size:           Program 68 bytes                                ;
;                       Data     5 bytes plus                           ;
;                                2 bytes in page zero                   ;
;                                                                       ;
;                                                                       ;

;PAGE ZERO POINTER
ARYPTR: .EQU    0D0H            ;PAGE ZERO POINTER TO THE ARRAY

MFILL:
        ;POP THE PARAMETERS FROM THE STACK
        PLA
```

```
        STA     RETADR
        PLA
        STA     RETADR+1        ;GET THE RETURN ADDRESS

        PLA
        STA     VALUE           ;GET FILL VALUE

        PLA
        STA     ARYSZ
        PLA
        STA     ARYSZ+1         ;GET SIZE OF AREA

        PLA
        STA     ARYPTR
        PLA
        STA     ARYPTR+1        ;GET STARTING ADDRESS OF AREA

        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA                     ;RESTORE RETURN ADDRESS

        ;
        ;DO THE FULL PAGES FIRST
        LDA     VALUE           ;GET VALUE FOR FILL
        LDX     ARYSZ+1         ;X = NUMBER OF PAGES TO DO
        BEQ     PARTPG          ;BRANCH IF THE HIGH BYTE OF SIZE = 0
        LDY     #0
FULLPG:
        STA     (ARYPTR),Y      ;STORE VALUE
        INY                     ;INCREMENT TO NEXT BYTE
        BNE     FULLPG          ;BRANCH IF NOT DONE WITH THIS PAGE
        INC     ARYPTR+1        ;ADVANCE TO THE NEXT PAGE
        DEX
        BNE     FULLPG          ;BRANCH IF NOT DONE WITH THE FULL PAGES

        ;
        ;DO THE REMAINING PARTIAL PAGE
        ; REGISTER A STILL CONTAINS VALUE
PARTPG:
        LDX     ARYSZ           ;GET THE NUMBER OF BYTES IN THIS FINAL PAGE
        BEQ     EXIT            ;BRANCH IF LOW BYTE OF SIZE = 0
        LDY     #0
PARTLP:
        STA     (ARYPTR),Y      ;STORE VALUE
        INY                     ;INCREMENT INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     PARTLP          ;BRANCH IF PARTIAL PAGE IS NOT DONE

EXIT:
        RTS

;DATA
ARYSZ:  .BLOCK  2               ;NUMBER OF BYTES TO INITIALIZE
VALUE:  .BLOCK  1               ;VALUE TO INITIALIZE ARRAY WITH
```

```
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS

;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION                                             ;
;                                                                    ;
;                                                                    ;

SC0501:
        ;FILL A SMALL BUFFER WITH 00
        LDA     BF1ADR+1
        PHA
        LDA     BF1ADR
        PHA                     ;PUSH STARTING ADDRESS
        LDA     BF1SZ+1
        PHA
        LDA     BF1SZ
        PHA                     ;PUSH NUMBER OF BYTES
        LDA     #0
        PHA                     ;PUSH VALUE
        JSR     MFILL           ;FILL BUFFER
        BRK

        ;FILL A BIG BUFFER WITH EA HEX (NOP)
        LDA     BF2ADR+1
        PHA
        LDA     BF2ADR
        PHA                     ;PUSH STARTING ADDRESS
        LDA     BF2SZ+1
        PHA
        LDA     BF2SZ
        PHA                     ;PUSH NUMBER OF BYTES
        LDA     #0EAH
        PHA                     ;PUSH VALUE
        JSR     MFILL           ;FILL BUFFER
        BRK
        JMP     SC0501


SIZE1:  .EQU    47H
SIZE2:  .EQU    6000H

BF1ADR: .WORD   BF1
BF2ADR: .WORD   BF2

BF1SZ:  .WORD   SIZE1
BF2SZ:  .WORD   SIZE2

BF1:    .BLOCK  SIZE1
BF2:    .BLOCK  SIZE2

        .END
```

Block Move (BLKMOV) 5B

Moves a block of data from a source area to a destination area.

Procedure: The program determines if the starting address of the destination area is within the source area. If it is, then working up from the starting address would overwrite some of the source data. To avoid that problem, the program works down from the highest address (this is sometimes called *move right*). If the starting address of the destination area is not within the source area, the program simply moves the data starting from the lowest address (this is sometimes called a *move left*). In either case, the program moves the data by handling complete pages separately from the remaining partial page. This approach allows the program to use 8-bit counters rather than a 16-bit counter, thus reducing execution time (although increasing memory usage). An area size (number of bytes to move) of 0000_{16} causes an exit with no memory changed.

Important Note: The user should be careful if either the source or the destination area includes the temporary storage used by the program itself. The program provides automatic address wraparound (mod 64K), but the results of any move involving the program's own temporary storage are unpredictable.

Registers Used: All

Execution Time: 128 cycles overhead plus the following:

1. If data can be moved starting from the lowest address (i.e., left):
   20 + 4110 * (more significant byte of number of bytes to move) + 18 * (less significant byte of number of bytes to move).

2. If data must be moved starting from the highest address (i.e., right) because of overlap:
   42 + 4622 * (more significant byte of number of bytes to move) + 18 * (less significant byte of number of bytes to move).

Program Size: 157 bytes

Data Memory Required: Two bytes anywhere in RAM for the length of the move (starting at address MVELEN), four bytes on page 0 for source and destination pointers (starting at addresses MVSRCE and MVDEST taken as addresses $00D0_{16}$ and $00D1_{16}$ — source pointer — and addresses $00D2_{16}$ and $00D3_{16}$ — destination pointer — in the listing).

Special Cases:

1. A size (number of bytes to move) of zero causes an immediate exit with no memory changed.

2. Moving data to or from areas occupied or used by the program itself will produce unpredictable results. Obviously, moving data to or from page 0 requires caution, since both this routine and most systems programs use that page. This routine does provide automatic address wraparound (mod 64K) for consistency, but the user must still approach moves involving page 0 carefully.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address
Less significant byte of number of bytes to move
More significant byte of number of bytes to move
Less significant byte of lowest address of destination area
More significant byte of lowest address of destination area
Less significant byte of lowest address of source area
More significant byte of lowest address of source area

Exit Conditions

The block of memory is moved from the source area to the destination area. If the number of bytes to be moved is NBYTES, the lowest address in the destination area is DEST, and the lowest address in the source area is SOURCE, then the area from addresses SOURCE through SOURCE + NBYTES − 1 is moved to addresses DEST through DEST + NBYTES − 1.

Examples

1. Data: Number of bytes to move = 0200_{16}
   Lowest address in destination area = $05D1_{16}$
   Lowest address in source area = $035E_{16}$
   Result: The contents of memory locations $035E_{16}$ through $055D_{16}$ are moved to $05D1_{16}$ through $07D0_{16}$.
2. Data: Number of bytes to move = $1B7A_{16}$
   Lowest address in destination area = $C946_{16}$
   Lowest address in source area = $C300_{16}$
   Result: The contents of memory locations $C300_{16}$ through $DE79_{16}$ are moved to $C946_{16}$ through $E4BF_{16}$

Note that Example 2 presents a more complex problem than Example 1 because the source and destination areas overlap. If, for instance, the program were simply to move data to the destination area starting from the lowest address, it would initially move the contents of $C300_{16}$ to $C946_{16}$. This would destroy the old contents of $C946_{16}$, which are needed later in the move. The solution to this problem is to move the data starting from the highest address if the destination area is above the source area but overlaps it.

```
;       Title           Block Move                                      ;
;       Name:           BLKMOV                                          ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Move data from source to destination            ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of number of bytes to move,          ;
;                         High byte of number of bytes to move,         ;
;                         Low byte of lowest address in destination     ;
;                           area,                                       ;
;                         High byte of lowest address in destination    ;
;                           area,                                       ;
;                         Low byte of lowest address in source area,    ;
;                         High byte of lowest address in source area    ;
;                                                                       ;
;       Exit:           Data moved from source to destination           ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           102 cycles overhead plus move                   ;
;                         move left cycles equals                       ;
;                               20 +                                    ;
;                               (high byte of length * 4110) +          ;
;                               (low byte of length * 18)               ;
;                                                                       ;
;                         move right cycles equals                      ;
;                               42 +                                    ;
;                               (high byte of length * 4622) +          ;
;                               (low byte of length * 18)               ;
;                                                                       ;
;       Size:           Program 146 bytes                               ;
;                       Data      2 bytes plus                          ;
;                                 4 bytes in page zero                  ;
;                                                                       ;
;                                                                       ;

;PAGE ZERO POINTERS
MVSRCE  .EQU    0D0H            ;SOURCE ADDRESS
MVDEST  .EQU    0D2H            ;DESTINATION ADDRESS

BLKMOV:
        ;GET RETURN ADDRESS
        PLA
        TAY                     ;SAVE LOW BYTE
        PLA
        TAX                     ;SAVE HIGH BYTE

        ;GET NUMBER OF BYTES
        PLA
```

```
        STA     MVELEN          ;STORE LOW BYTE
        PLA
        STA     MVELEN+1        ;STORE HIGH BYTE

        ;GET STARTING DESTINATION ADDRESS
        PLA
        STA     MVDEST          ;STORE LOW BYTE
        PLA
        STA     MVDEST+1        ;STORE HIGH BYTE

        ;GET STARTING SOURCE ADDRESS
        PLA
        STA     MVSRCE          ;STORE LOW BYTE
        PLA
        STA     MVSRCE+1        ;STORE HIGH BYTE

        ;RESTORE RETURN ADDRESS
        TXA
        PHA                     ;RESTORE HIGH BYTE
        TYA
        PHA                     ;RESTORE LOW BYTE


        ;
        ;DETERMINE IF DESTINATION AREA IS ABOVE SOURCE AREA BUT OVERLAPS
        ; IT. REMEMBER, OVERLAP CAN BE MOD 64K. OVERLAP OCCURS IF
        ; STARTING DESTINATION ADDRESS MINUS STARTING SOURCE ADDRESS (MOD 64K)
        ; IS LESS THAN NUMBER OF BYTES TO MOVE
        LDA     MVDEST          ;CALCULATE DESTINATION - SOURCE
        SEC
        SBC     MVSRCE
        TAX
        LDA     MVDEST+1
        SBC     MVSRCE+1        ;MOD 64K IS AUTOMATIC - DISCARD CARRY
        TAY
        TXA                     ;COMPARE WITH NUMBER OF BYTES TO MOVE
        CMP     MVELEN
        TYA
        SBC     MVELEN+1
        BCS     DOLEFT          ;BRANCH IF NO PROBLEM WITH OVERLAP


        ;DESTINATION AREA IS ABOVE SOURCE AREA BUT OVERLAPS IT
        ;MOVE FROM HIGHEST ADDRESS TO AVOID DESTROYING DATA
        JSR     MVERHT
        JMP     EXIT

        ;NO PROBLEM DOING ORDINARY MOVE STARTING AT LOWEST ADDRESS
DOLEFT:
        JSR     MVELFT


EXIT:
        RTS
```

```
;*****************************************
;SUBROUTINE: MVELFT
;PURPOSE:  MOVE SOURCE TO DESTINATION STARTING FROM
;       THE LOWEST ADDRESS
;ENTRY: MVSRCE  = 2 BYTE LOWEST ADDRESS OF SOURCE AREA
;       MVDEST  = 2 BYTE LOWEST ADDRESS OF DESTINATION AREA
;       MVELEN  = 2 BYTE NUMBER OF BYTES TO MOVE
;EXIT:  SOURCE MOVED TO DESTINATION
;*****************************************

MVELFT:
        LDY     #0              ;ZERO INDEX
        LDX     MVELEN+1        ;X= NUMBER OF FULL PAGES TO MOVE
        BEQ     MLPART          ;IF X = 0 THEN DO PARTIAL PAGE
MLPAGE:
        LDA     (MVSRCE),Y
        STA     (MVDEST),Y      ;MOVE ONE BYTE
        INY                     ;NEXT BYTE
        BNE     MLPAGE          ;CONTINUE UNTIL 256 BYTES ARE MOVED
        INC     MVSRCE+1        ;ADVANCE TO NEXT PAGE OF SOURCE
        INC     MVDEST+1        ; AND DESTINATION
        DEX                     ;DECREMENT PAGE COUNT
        BNE     MLPAGE          ;CONTINUE UNTIL ALL FULL PAGES ARE MOVED

MLPART:
        LDX     MVELEN          ;GET LENGTH OF LAST PAGE
        BEQ     MLEXIT          ;BRANCH IF LENGTH OF LAST PAGE = 0
                                ;REGISTER Y IS 0
MLLAST:
        LDA     (MVSRCE),Y
        STA     (MVDEST),Y      ;MOVE BYTE
        INY                     ;NEXT BYTE
        DEX                     ;DECREMENT COUNTER
        BNE     MLLAST          ;CONTINUE UNTIL LAST PAGE IS DONE

MLEXIT:
        RTS

;*****************************************
;SUBROUTINE: MVERHT
;PURPOSE:  MOVE SOURCE TO DESTINATION STARTING FROM
;       THE HIGHEST ADDRESS
;ENTRY: MVSRCE  = 2 BYTE LOWEST ADDRESS OF SOURCE AREA
;       MVDEST  = 2 BYTE LOWEST ADDRESS OF DESTINATION AREA
;       MVELEN  = 2 BYTE NUMBER OF BYTES TO MOVE
;EXIT:  SOURCE MOVED TO DESTINATION
;*****************************************

MVERHT:
        ;

        ;MOVE THE PARTIAL PAGE FIRST
        LDA     MVELEN+1
        CLC
        ADC     MVSRCE+1
        STA     MVSRCE+1        ;POINT TO LAST PAGE OF SOURCE
```

```
        LDA     MVELEN+1
        CLC
        ADC     MVDEST+1
        STA     MVDEST+1        ;POINT TO LAST PAGE OF DESTINATION

        ;MOVE THE LAST PARTIAL PAGE FIRST
        LDY     MVELEN          ;GET LENGTH OF LAST PAGE
        BEQ     MRPAGE          ;IF Y = 0 THEN DO THE FULL PAGES

MR0:
        DEY                     ;BACK UP Y TO THE NEXT BYTE
        LDA     (MVSRCE),Y
        STA     (MVDEST),Y      ;MOVE BYTE
        CPY     #0
        BNE     MR0             ;BRANCH IF NOT DONE WITH THE LAST PAGE

MRPAGE:
        LDX     MVELEN+1        ;GET HIGH BYTE OF COUNT AS PAGE COUNTER
        BEQ     MREXIT          ;BRANCH IF HIGH BYTE = 0 (NO FULL PAGES)
MR1:
        DEC     MVSRCE+1        ;BACK UP TO PREVIOUS PAGE OF SOURCE
        DEC     MVDEST+1        ; AND DESTINATION
MR2:
        DEY                     ;BACK UP Y TO THE NEXT BYTE
        LDA     (MVSRCE),Y
        STA     (MVDEST),Y      ;MOVE BYTE
        CPY     #0
        BNE     MR2             ;BRANCH IF NOT DONE WITH THIS PAGE
        DEX                     ;DECREMENT PAGE COUNTER
        BNE     MR1             ;BRANCH IF NOT ALL PAGES ARE MOVED

MREXIT:
        RTS

;
;DATA SECTION
MVELEN  .BLOCK  2               ;LENGTH OF MOVE

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION: MOVE 0800 THROUGH 097F TO 0900 THROUGH 0A7F   ;
;                                                                       ;
;                                                                       ;


SC0502:
        LDA     SRCE+1
        PHA                     ;PUSH HIGH BYTE OF SOURCE
        LDA     SRCE
        PHA                     ;PUSH LOW BYTE OF SOURCE
        LDA     DEST+1
        PHA                     ;PUSH HIGH BYTE OF DESTINATION
        LDA     DEST
        PHA                     ;PUSH LOW BYTE OF DESTINATION
```

```
        LDA     LEN+1
        PHA                     ;PUSH HIGH BYTE OF LENGTH
        LDA     LEN
        PHA                     ;PUSH LOW BYTE OF LENGTH
        JSR     BLKMOV          ;MOVE DATA FROM SOURCE TO DESTINATION
        BRK                     ; FOR THE DEFAULT VALUES MEMORY FROM 800 HEX
                                ; THROUGH 97F HEX IS MOVED TO 900 HEX THROUGH
                                ; A7F HEX.
        JMP     SC0502

;
;TEST DATA, CHANGE TO TEST OTHER VALUES
SRCE    .WORD   0800H           ;STARTING ADDRESS OF SOURCE AREA
DEST    .WORD   0900H           ;STARTING ADDRESS OF DESTINATION AREA
LEN     .WORD   0180H           ;NUMBER OF BYTES TO MOVE

        .END    ;PROGRAM
```

One-Dimensional Byte Array Index (D1BYTE) 5C

Calculates the address of an element of a byte-length array, given the base address and the subscript (index) of the element.

Procedure: The program simply adds the base address to the subscript. The sum is the address of the element.

Registers Used: All

Execution Time: 74 cycles

Program Size: 37 bytes

Date Memory Required: Four bytes anywhere in RAM to hold the return address (two bytes starting at address RETADR) and the subscript (two bytes starting at address SUBSCR).

Entry Conditions

Order in steck (starting et the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of subscript
More significant byte of subscript

Less significant byte of base address of array
More significant byte of base address of array

Exit Conditions

(A) = More significant byte of address of element

(Y) = Less significant byte of address of element

Examples

1. Data: Base address = $0E00_{16}$
   Subscript = $012C_{16}$

   Result: Address of element = $0E00_{16} + 012C_{16} = 0F2C_{16}$.

2. Data: Base address = $C4E1_{16}$
   Subscript = $02E4_{16}$

   Result: Address of element = $C4E1_{16} + 02E4_{15} = C7C5_{16}$.

```
;       Title           One dimensional byte array indexing             ;
;       Name:           D1BYTE                                          ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Given the base address of a byte array and a    ;
;                       subscript 'I' calculate the address of A[I]     ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of subscript,                        ;
;                         High byte of subscript,                       ;
;                         Low byte of base address of array,            ;
;                         High byte of base address of array            ;
;                                                                       ;
;       Exit:           Register A = High byte of address               ;
;                       Register Y = Low byte of address                ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           74 cycles                                       ;
;                                                                       ;
;       Size:           Program 37 bytes                                ;
;                       Data     4 bytes                                ;
;                                                                       ;
;                                                                       ;

D1BYTE:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET SUBSCRIPT
        PLA
        STA     SS
        PLA
        STA     SS+1

        ;ADD BASE ADDRESS TO SUBSCRIPT
        PLA
        CLC
        ADC     SS
        TAY                     ;REGISTER Y = LOW BYTE
        PLA
        ADC     SS+1
        TAX                     ;SAVE HIGH BYTE IN REGISTER X

        ;RESTORE RETURN ADDRESS TO STACK
        LDA     RETADR+1
        PHA
```

```
        LDA     RETADR
        PHA                             ;RESTORE RETURN ADDRESS

        TXA                             ;GET HIGH BYTE BACK TO REGISTER A
        RTS                             ;EXIT


;
;DATA
RETADR: .BLOCK  2                       ;TEMPORARY FOR RETURN ADDRESS
SS:     .BLOCK  2                       ;SUBSCRIPT INTO THE ARRAY


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0503:
        ;PUSH ARRAY ADDRESS
        LDA     ARYADR+1                ;HIGH BYTE
        PHA
        LDA     ARYADR                  ;LOW BYTE
        PHA

        ;PUSH A SUBSCRIPT
        LDA     SUBSCR+1                ;HIGH BYTE
        PHA
        LDA     SUBSCR                  ;LOW BYTE
        PHA
        JSR     DlBYTE                  ;CALCULATE ADDRESS
        BRK                             ;AY = ARY+2
                                        ;   = ADDRESS OF ARY(2), WHICH CONTAINS 3

        JMP     SC0503

;
;TEST DATA, CHANGE SUBSCR FOR OTHER VALUES
SUBSCR: .WORD   2                       ;TEST SUBSCRIPT INTO THE ARRAY
ARYADR: .WORD   ARY                     ;BASE ADDRESS OF ARRAY

;THE ARRAY (8 ENTRIES)
ARY:    .BYTE   1,2,3,4,5,6,7,8

        .END    ;PROGRAM
```

One-Dimensional Word Array Index (D1WORD) 5D

Calculates the starting address of an element of a word-length (16-bit) array, given the base address of the array and the subscript (index) of the element. The element occupies the starting address and the address one larger; elements may be organized with either the less significant byte or the more significant byte in the starting address.

Procedure: The program multiplies the subscript by two (using a logical left shift) before adding it to the base address. The sum (BASE + 2*SUBSCRIPT) is then the starting address of the element.

Registers Used: All

Execution Time: 78 cycles

Program Size: 39 bytes

Data Memory Required: Four bytes anywhere in RAM to hold the return address (two bytes starting at address RETADR) and the subscript (two bytes starting at address SUBSCR).

Entry Conditions

Order in stack (starting at the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of subscript
More significant byte of subscript

Less significant byte of base address of array
More significant byte of base address of array

Exit Conditions

(A) = More significant byte of starting address of element

(Y) = Less significant byte of starting address of element

Examples

1. Data: Base address = $A148_{16}$
   Subscript = $01A9_{16}$

   Result: Address of first byte of element
   $= A148_{16} + 2 \times 01A9_{16}$
   $= A148_{16} + 0342_{16} = A49A_{16}$.
   That is, the word-length element occupies addresses $A49A_{16}$ and $A49B_{16}$.

2. Data: Base address = $C4E0_{16}$
   Subscript = $015B_{16}$

   Result: Address of first byte of element
   $= C4E0_{16} + 2 \times 015B_{16}$
   $= C4E0_{16} + 02B6_{16} = C796_{16}$.
   That is, the word-length element occupies addresses $C796_{16}$ and $C797_{16}$.

```
;       Title           One dimensional word array indexing
;       Name:           D1WORD
;
;
;
;       Purpose:        Given the base address of a word array and a
;                       subscript 'I' calculate the address of A[I]
;
;       Entry:          TOP OF STACK
;                         Low byte of return address,
;                         High byte of return address,
;                         Low byte of subscript,
;                         High byte of subscript,
;                         Low byte of base address of array,
;                         High byte of base address of array
;
;       Exit:           Register A = High byte of address
;                       Register Y = Low byte of address
;
;       Registers used: All
;
;       Time:           78 cycles
;
;       Size:           Program 39 bytes
;                       Data     4 bytes
;
;

D1WORD:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET SUBSCRIPT AND MULTIPLY IT BY 2
        PLA
        ASL     A
        STA     SS
        PLA
        ROL     A
        STA     SS+1

        ;ADD BASE ADDRESS TO DOUBLED SUBSCRIPT
        PLA
        CLC
        ADC     SS
        TAY                     ;REGISTER Y = LOW BYTE
        PLA
        ADC     SS+1
        TAX                     ;SAVE HIGH BYTE IN REGISTER X

        ;RESTORE RETURN ADDRESS TO STACK
```

```
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA                     ;RESTORE RETURN ADDRESS

        TXA                     ;GET HIGH BYTE BACK TO REGISTER A
        RTS                     ;EXIT

;
;DATA
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS
SS:     .BLOCK  2               ;SUBSCRIPT INTO THE ARRAY

;
;
;       SAMPLE EXECUTION:
;
;

SC0504:
        ;PUSH ARRAY ADDRESS
        LDA     ARYADR+1
        PHA
        LDA     ARYADR
        PHA

        ;PUSH A SUBSCRIPT OF 3
        LDA     SUBSCR+1
        PHA
        LDA     SUBSCR
        PHA
        JSR     D1WORD          ;CALCULATE ADDRESS
        BRK                     ;FOR THE INITIAL TEST DATA
                                ;AY = STARTING ADDRESS OF ARY(3)
                                ;   = ARY + (3*2)
                                ;   = ARY + 6
                                ;   = ARY(3) CONTAINS 240 HEX
        JMP     SC0504

;
;TEST DATA
SUBSCR: .WORD   3               ;TEST SUBSCRIPT INTO ARY
ARYADR: .WORD   ARY             ;BASE ADDRESS OF ARRAY

;THE ARRAY (8 ENTRIES)
ARY:    .WORD   0180H,01C0H,0200H,0240H,0280H,02C0H,03E7H,0A34H
        .END    ;PROGRAM
```

Two-Dimensional Byte Array Index (D2BYTE) 5E

Calculates the address of an element of a two-dimensional byte-length array, given the base address of the array, the two subscripts of the element, and the size of a row (that is, the number of columns). The array is assumed to be stored in row major order (that is, by rows) and both subscripts are assumed to begin at zero.

Procedure: The program multiplies the row size (number of columns in a row) times the row subscript (since the elements are stored by rows) and adds the product to the column subscript. It then adds the sum to the base address. The program performs the multiplication using a standard shift-and-add algorithm (see Subroutine 6H).

Registers Used: All

Execution Time: Approximately 1500 cycles, depending mainly on the amount of time required to perform the multiplication.

Program Size: 119 bytes

Data Memory Required: Ten bytes anywhere in memory to hold the return address (two bytes starting at address RETADR), the row subscript (two bytes starting at address SS1), the size (length) of the rows (two bytes starting at address SS1SZ), the column subscript (two bytes starting at address SS2), and the product of row size times row subscript (two bytes starting at address PROD).

Entry Conditions

Order in stack (starting at the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of column subscript
More significant byte of column subscript

Less significant byte of the size of a row
More significant byte of the size of a row

Less significant byte of row subscript
More significant byte of row subscript

Less significant byte of bass address of array
More significant byte of base address of array

Exit Conditions

(A) = More significant byte of address of element
(Y) = Less significant byte of address of element

Examples

1. Data: Base address = $3C00_{16}$
Column subscript = 0004_{16}
Size of row (number of columns) = 0018_{16}
Row subscript = 0003_{16}

   Result: Address of element = $3C00_{16} + 0003_{16} \times 0018_{16} + 0004_{16}$
$= 3C00_{16} + 0048_{16} + 0004_{16}$
$= 3C4C_{16}$.
Thus the address of ARRAY (3,4) is $3C4C_{16}$.

2. Data: Base address = $6A4A_{16}$
Column subscript = 0035_{16}
Size of row (number of columns) = 0050_{16}
Row subscript = 0002_{16}

   Result: Address of element = $6A4A_{16} + 0002_{16} \times 0050_{16} + 0035_{16} = 6A4A_{16} + 00A0_{16} + 0035_{16} = 6B1F_{16}$.
Thus the address of ARRAY (2,35) is $6B1F_{16}$.

The general formula is
ADDRESS OF ELEMENT = BASE ADDRESS OF ARRAY + ROW SUBSCRIPT × SIZE OF ROW + COLUMN SUBSCRIPT

Note that we refer to the *size* of the row subscript; the size is the number of consecutive memory addresses for which the subscript has the same value. This is also the number of bytes from the starting address of an element to the starting address of the element with the same column subscript but a row subscript one larger.

```
;       Title           Two dimensional byte array indexing                 ;
;       Name:           D2BYTE                                              ;
;                                                                           ;
;                                                                           ;
;                                                                           ;
;       Purpose:        Given the base address of a byte array, two         ;
;                       subscripts 'I','J', and the size of the first       ;
;                       subscript in bytes, calculate the address of        ;
;                       A[I,J]. The array is assumed to be stored in        ;
;                       row major order (A[0,0], A[0,1],..., A[K,L]),       ;
;                       and both dimensions are assumed to begin at         ;
;                       zero as in the following Pascal declaration:        ;
;                         A:ARRAY[0..2,0..7] OF BYTE;                       ;
;                                                                           ;
;       Entry:          TOP OF STACK                                        ;
;                         Low byte of return address,                       ;
;                         High byte of return address,                      ;
;                         Low byte of second subscript,                     ;
;                         High byte of second subscript,                    ;
;                         Low byte of size of first subscript in bytes,     ;
```

```
;                       High byte of size of first subscript in bytes,;
;                       Low byte of first subscript,                  ;
;                       High byte of first subscript,                 ;
;                       Low byte of base address of array,            ;
;                       High byte of base address of array            ;
;                     NOTE:                                           ;
;                       The size of the first subscript is the length ;
;                       of a row                                      ;
;                                                                     ;
;       Exit:           Register A = High byte of address             ;
;                       Register Y = Low byte of address              ;
;                                                                     ;
;       Registers used: All                                           ;
;                                                                     ;
;       Time:           Approximately 1500 cycles                     ;
;                                                                     ;
;       Size:           Program 119 bytes                             ;
;                       Data     10 bytes                             ;
;                                                                     ;
;                                                                     ;

D2BYTE:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET SECOND SUBSCRIPT
        PLA
        STA     SS2
        PLA
        STA     SS2+1

        ;GET SIZE OF FIRST SUBSCRIPT (LENGTH OF A ROW)
        PLA
        STA     SS1SZ
        PLA
        STA     SS1SZ+1

        ;GET FIRST SUBSCRIPT
        PLA
        STA     SS1
        PLA
        STA     SS1+1

        ;MULTIPLY FIRST SUBSCRIPT * ROW LENGTH USING THE SHIFT AND ADD
        ; ALGORITHM. THE RESULT WILL BE IN SS1
        LDA     #0              ;PARTIAL PRODUCT = ZERO INITIALLY
        STA     PROD
        STA     PROD+1
        LDX     #17             ;NUMBER OF SHIFTS = 17
        CLC
```

```
MULLP:
        ROR     PROD+1          ;SHIFT PARTIAL PRODUCT
        ROR     PROD
        ROR     SS1+1           ;SHIFT MULTIPLIER
        ROR     SS1
        BCC     DECCNT
        CLC                     ;ADD MULTIPLICAND TO PARTIAL PRODUCT
        LDA     SS1SZ           ; IF NEXT BIT OF MULTIPLIER IS 1
        ADC     PROD
        STA     PROD
        LDA     SS1SZ+1
        ADC     PROD+1
        STA     PROD+1

DECCNT:
        DEX
        BNE     MULLP

        ;ADD IN THE SECOND SUBSCRIPT
        LDA     SS1
        CLC
        ADC     SS2
        STA     SS1
        LDA     SS1+1
        ADC     SS2+1
        STA     SS1+1

        ;ADD BASE ADDRESS TO FORM THE FINAL ADDRESS
        PLA
        CLC
        ADC     SS1
        TAY                     ;REGISTER Y = LOW BYTE
        PLA
        ADC     SS1+1
        TAX                     ;SAVE HIGH BYTE IN REGISTER X

        ;RESTORE RETURN ADDRESS TO STACK
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA                     ;RESTORE RETURN ADDRESS

        TXA                     ;GET HIGH BYTE BACK TO REGISTER A
        RTS                     ;EXIT


;
;DATA
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS
SS1:    .BLOCK  2               ;SUBSCRIPT 1
SS1SZ:  .BLOCK  2               ;SIZE OF SUBSCRIPT 1 IN BYTES
SS2:    .BLOCK  2               ;SUBSCRIPT 2
PROD:   .BLOCK  2               ;TEMPORARY FOR THE MULTIPLY
```

```
;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION:                                            ;
;                                                                    ;
;                                                                    ;

SC0505:
        ;PUSH ARRAY ADDRESS
        LDA     ARYADR+1
        PHA
        LDA     ARYADR
        PHA

        ;PUSH FIRST SUBSCRIPT
        LDA     SUBS1+1
        PHA
        LDA     SUBS1
        PHA

        ;PUSH SIZE OF FIRST SUBSCRIPT
        LDA     SSUBS1+1
        PHA
        LDA     SSUBS1
        PHA

        ;PUSH SECOND SUBSCRIPT
        LDA     SUBS2+1
        PHA
        LDA     SUBS2
        PHA

        JSR     D2BYTE          ;CALCULATE ADDRESS
        BRK                     ;FOR THE INITIAL TEST DATA
                                ;AY = ADDRESS OF ARY(2,4)
                                ;    = ARY + (2*8) + 4
                                ;    = ARY + 20 (CONTENTS ARE 21)

        JMP     SC0505

;
;DATA
SUBS1:   .WORD   2               ;SUBSCRIPT 1
SSUBS1:  .WORD   8               ;SIZE OF SUBSCRIPT 1
SUBS2:   .WORD   4               ;SUBSCRIPT 2
ARYADR:  .WORD   ARY             ;ADDRESS OF ARRAY

;THE ARRAY (3 ROWS OF 8 COLUMNS)
ARY:     .BYTE   1 ,2 ,3 ,4 ,5 ,6 ,7 ,8
         .BYTE   9 ,10,11,12,13,14,15,16
         .BYTE   17,18,19,20,21,22,23,24

         .END    ;PROGRAM
```

Two-Dimensional Word Array Index (D2WORD) 5F

Calculates the starting address of an element of a two-dimensional word-length (16-bit) array, given the base address of the array, the two subscripts of the element, and the size of a row in bytes. The array is assumed to be stored in row major order (that is, by rows) and both subscripts are assumed to begin at zero.

Procedure: The program multiplies the row size (in bytes) times the row subscript (since the elements are stored by row), adds the product to the doubled column subscript (doubled because each element occupies two bytes), and adds the sum to the base address. The program uses a standard shift-and-add algorithm (see Subroutine 6H) to multiply.

Registers Used: All

Execution Time: Approximately 1500 cycles, depending mainly on the amount of time required to perform the multiplication of row size in bytes times row subscript.

Program Size: 121 bytes

Data Memory Required: Ten bytes anywhere in memory to hold the return address (two bytes starting at address RETADR), the row subscript (two bytes starting at address SS1), the row size in bytes (two bytes starting at address SS1SZ), the column subscript (two bytes starting at address SS2), and the product of row size times row subscript (two bytes starting at address PROD).

Entry Conditions

Order in stack (starting at the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of column subscript
More significant byte of column subscript

Less significant byte of size of rows (in bytes)
More significant byte of size of rows (in bytes)

Less significant byte of row subscript
More significant byte of row subscript

Less significant byte of base address of array
More significant byte of base address of array

Exit Conditions

(A) = More significant byte of starting address of element
(Y) = Less significant byte of starting address of element
The element occupies the address in AY and the next higher address.

Examples

1. Data: Base address = $5E14_{16}$
   Column subscript = 0008_{16}
   Size of a row (in bytes) = $001C_{16}$
   (i.e., each row has 0014_{10} or $000E_{16}$ word-length elements)
   Row subscript = 0005_{16}

   Result: Starting address of element
   $= 5E14_{16} + 0005_{16} \times 001C_{16} + 0008_{16} \times 2 = 5E14_{16} + 008C_{16} + 0010_{16} = 5EB0_{16}$.
   Thus, the starting address of ARRAY (5,8) is $5EB0_{16}$ and the element occupies addresses $5EB0_{16}$ and $5EB1_{16}$.

2. Data: Base address = $B100_{16}$
   Column subscript = 0002_{16}
   Size of a row (in bytes) = 0008_{16}
   (i.e., each row has 4 word-length elements)
   Row subscript = 0006_{16}

   Result: Starting address of element
   $= B100_{16} + 0006_{16} \times 0008_{16} + 0002_{16} \times 2 = B100_{16} + 0030_{16} + 0004_{16} = B134_{16}$.
   Thus, the starting address of ARRAY (6,2) is $B134_{16}$ and the element occupies addresses $B134_{16}$ and $B135_{16}$.

The general formula is
STARTING ADDRESS OF ELEMENT
= BASE ADDRESS OF ARRAY
+ ROW SUBSCRIPT × SIZE OF ROW
+ COLUMN SUBSCRIPT × 2

Note that one parameter of this routine is the size of a row in bytes. The size in the case of word-length elements is the number of columns (per row) times two (the size of an element). The reason why we chose this parameter rather than the number of columns or the maximum column index is that this parameter can be calculated once (when the array bounds are determined) and used whenever the array is accessed. The alternative parameters (number of columns or maximum column index) would require extra calculations as part of each indexing operation.

```
;       Title           Two dimensional word array indexing                 ;
;       Name:           D2WORD                                              ;
;                                                                           ;
;                                                                           ;
;                                                                           ;
;       Purpose:        Given the base address of a word array, two         ;
;                       subscripts 'I','J', and the size of the first       ;
;                       subscript in bytes, calculate the address of        ;
;                       A[I,J]. The array is assumed to be stored in        ;
;                       row major order (A[0,0], A[0,1],..., A[K,L]),       ;
```

```
;                       and both dimensions are assumed to begin at      ;
;                       zero as in the following Pascal declaration:     ;
;                         A:ARRAY[0..2,0..7] OF WORD;                    ;
;                                                                        ;
;       Entry:          TOP OF STACK                                     ;
;                         Low byte of return address,                    ;
;                         High byte of return address,                   ;
;                         Low byte of second subscript,                  ;
;                         High byte of second subscript,                 ;
;                         Low byte of size of first subscript in bytes,  ;
;                         High byte of size of first subscript in bytes, ;
;                         Low byte of first subscript,                   ;
;                         High byte of first subscript,                  ;
;                         Low byte of base address of array,             ;
;                         High byte of base address of array             ;
;                                                                        ;
;       Exit:           Register A = High byte of address                ;
;                       Register Y = Low byte of address                 ;
;                                                                        ;
;       Registers used: ALL                                              ;
;                                                                        ;
;       Time:           Approximately 1500 cycles                        ;
;                                                                        ;
;       Size:           Program 121 bytes                                ;
;                       Data     10 bytes                                ;
;                                                                        ;
;                                                                        ;

D2WORD:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET SECOND SUBSCRIPT AND MULTIPLY BY 2 FOR WORD-LENGTH ELEMENTS
        PLA
        ASL     A
        STA     SS2
        PLA
        ROL     A
        STA     SS2+1

        ;GET SIZE OF FIRST SUBSCRIPT
        PLA
        STA     SS1SZ
        PLA
        STA     SS1SZ+1

        ;GET FIRST SUBSCRIPT
        PLA
        STA     SS1
        PLA
        STA     SS1+1
```

```
        ;MULTIPLY FIRST SUBSCRIPT * ROW SIZE (IN BYTES) USING THE SHIFT AND ADD
        ;ALGORITHM. THE RESULT WILL BE IN SS1
        LDA     #0              ;PARTIAL PRODUCT = ZERO INITIALLY
        STA     PROD
        STA     PROD+1
        LDX     #17             ;NUMBER OF SHIFTS = 17
        CLC
MULLP:
        ROR     PROD+1          ;SHIFT PARTIAL PRODUCT
        ROR     PROD
        ROR     SS1+1           ;SHIFT MULTIPLIER
        ROR     SS1
        BCC     DECCNT
        CLC                     ;ADD MULTIPLICAND TO PARTIAL PRODUCT
        LDA     SS1SZ           ; IF NEXT BIT OF MULTIPLIER IS 1
        ADC     PROD
        STA     PROD
        LDA     SS1SZ+1
        ADC     PROD+1
        STA     PROD+1

DECCNT:
        DEX
        BNE     MULLP

        ;ADD IN THE SECOND SUBSCRIPT DOUBLED
        LDA     SS1
        CLC
        ADC     SS2
        STA     SS1
        LDA     SS1+1
        ADC     SS2+1
        STA     SS1+1

        ;ADD BASE ADDRESS TO FORM THE FINAL ADDRESS
        PLA
        CLC
        ADC     SS1
        TAY                     ;REGISTER Y = LOW BYTE
        PLA
        ADC     SS1+1
        TAX                     ;SAVE HIGH BYTE IN REGISTER X

        ;RESTORE RETURN ADDRESS TO STACK
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA                     ;RESTORE RETURN ADDRESS

        TXA                     ;GET HIGH BYTE BACK TO REGISTER A
        RTS                     ;EXIT


;
;DATA
```

```
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS
SS1:    .BLOCK  2               ;SUBSCRIPT 1
SS1SZ:  .BLOCK  2               ;SIZE OF SUBSCRIPT 1 IN BYTES
SS2:    .BLOCK  2               ;SUBSCRIPT 2
PROD:   .BLOCK  2               ;TEMPORARY FOR THE MULTIPLY

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0506:
        ;PUSH ARRAY ADDRESS
        LDA     ARYADR+1
        PHA
        LDA     ARYADR
        PHA

        ;PUSH FIRST SUBSCRIPT
        LDA     SUBS1+1
        PHA
        LDA     SUBS1
        PHA

        ;PUSH SIZE OF FIRST SUBSCRIPT
        LDA     SSUBS1+1
        PHA
        LDA     SSUBS1
        PHA

        ;PUSH SECOND SUBSCRIPT
        LDA     SUBS2+1
        PHA
        LDA     SUBS2
        PHA

        JSR     D2WORD          ;CALCULATE ADDRESS
        BRK                     ;FOR THE INITIAL TEST DATA
                                ;AY = STARTING ADDRESS OF ARY(2,4)
                                ;   = ARY + (2*16) + (4*2)
                                ;   = ARY + 40
                                ;   = ARY(2,4) CONTAINS 2100 HEX
        JMP     SC0506

;
;DATA
SUBS1:  .WORD   2               ;SUBSCRIPT 1
SSUBS1: .WORD   16              ;SIZE OF SUBSCRIPT 1
SUBS2:  .WORD   4               ;SUBSCRIPT 2
ARYADR: .WORD   ARY             ;ADDRESS OF ARRAY

;THE ARRAY (3 ROWS OF 8 COLUMNS)
```

```
ARY:    .WORD   0100H,0200H,0300H,0400H,0500H,0600H,0700H,0800H
        .WORD   0900H,1000H,1100H,1200H,1300H,1400H,1500H,1600H
        .WORD   1700H,1800H,1900H,2000H,2100H,2200H,2300H,2400H

        .END    ;PROGRAM
```

N-Dimensional Array Index (NDIM) 5G

Calculates the starting address of an element of an N-dimensional array given the base address and N pairs of sizes and subscripts. The size of a dimension is the number of bytes from the starting address of an element to the starting address of the element with an index one larger in the dimension but the same in all other dimensions. The array is assumed to be stored in row major order (that is, organized so that subscripts to the right change before subscripts to the left).

Note that the size of the rightmost subscript is simply the size of the elements (in bytes); the size of the next subscript is the size of the elements times the maximum value of the rightmost subscript plus 1, etc. All subscripts are assumed to begin at zero; otherwise, the user must normalize the subscripts (see the second example at the end of the listing).

Procedure: The program loops on each dimension, calculating the offset in that dimension as the subscript times the size. If the size is an easy case (an integral power of 2), the program reduces the multiplication to left shifts. Otherwise, it performs each multiplication using the shift-and-add algorithm of Subroutine 6H. Once the program has calculated the overall offset, it adds that offset to the base address to obtain the starting address of the element.

Registers Used: All

Execution Time: Approximately 1100 cycles per dimension plus 90 cycles overhead. Depends mainly on the time required to perform the multiplications.

Program Size: 192 bytes

Data Memory Required: Eleven bytes anywhere in memory to hold the return address (two bytes starting at address RETADR), the current subscript (two bytes starting at address SS), the current size (two bytes starting at address SIZE), the accumulated offset (two bytes starting at address OFFSET), the number of dimensions (one byte at address NUMDIM), and the product of size times subscript (two bytes starting at address PROD).

Special Case: If the number of dimensions is zero, the program returns with the base address in registers A (more significant byte) and Y (less significant byte).

Entry Conditions

Order in stack (starting at the top)

Less significant byte of return address
More significant byte of return address

Number of dimensions

Less significant byte of size of rightmost dimension
More significant byte of size of rightmost dimension

Less significant byte of rightmost subscript
More significant byte of rightmost subscript

.
.
.

Less significant byte of size of leftmost dimension
More significant byte of size of leftmost dimension

Less significant byte of leftmost subscript
More significant byte of leftmost subscript

Less significant byte of base address of array
More significant byte of base address of array

Exit Conditions

(A) = More significant byte of address of element
(Y) = Less significant byte of address of element

The element occupies memory addresses from the calculated starting address through that address plus the rightmost subscript minus 1. That is, the element occupies memory addresses START through START + SIZE − 1, where START is the calculated address and SIZE is the size of an element in bytes.

Example

Data: Base address = $3C00_{16}$
Number of dimensions = 03_{16}
Rightmost subscript = 0005_{16}
Rightmost size = 0003_{16} (3-byte entries)
Middle subscript = 0003_{16}
Middle size = 0012_{16} (six 3-byte entries)
Leftmost subscript = 0004_{16}
Leftmost size = $007E_{16}$ (seven sets of six 3-byte entries)

Result: Address of entry = $3C00_{16} + 0005_{16} \times 0003_{16} + 0003_{16} \times 0012_{16} + 0004_{16} \times 007E_{16} = 3C00_{16} + 000F_{16} + 0036_{16} + 01F8_{16} = 3E3D_{16}$.
That is, the element is ARRAY (4,3,5); it occupies addresses $3E3D_{16}$ through $3E3F_{16}$. The maximum values of the various subscripts are 6 (leftmost) and 5 (middle). Each element consists of three bytes.

The general formula is

$$\text{STARTING ADDRESS} = \text{BASE ADDRESS} + \sum_{i=0}^{N-1} \text{SUBSCRIPT}_i \times \text{SIZE}_i$$

where:
N is the number of dimensions
SUBSCRIPT_i is the ith subscript
SIZE_i is the size of the ith dimension

Note that we use the sizes of each dimension as parameters to reduce the number of repetitive multiplications and to generalize the procedure. The sizes can be calculated (and saved) as soon as the bounds of the array are known. Those sizes can then be used whenever indexing is performed on that array. Obviously, the sizes do not change if the bounds are fixed and they should not be recalculated as part of each indexing operation. The sizes are also general, since the elements can themselves consist of any number of bytes.

```
;       Title           N dimensional array_indexing                    ;
;       Name:           NDIM                                            ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;
;       Purpose:        Calculate the address of an element in a        ;
;                       N dimensional array given the base address,     ;
;                       N pairs of size in bytes and subscript, and the ;
;                       number of dimensions of the array. The array is ;
;                       assumed to be stored in row major order         ;
;                       (A[0,0,0],A[0,0,1],...,A[0,1,0],A[0,1,1],...    ;
;                       Also it is assumed that all dimensions begin    ;
;                       at 0 as in the following Pascal declaration:    ;
;                         A:ARRAY[0..10,0..3,0..5] OF SOMETHING         ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Number of dimensions,                         ;
;                         Low byte of size (dim N-1) in bytes,          ;
;                         High byte of size (dim N-1) in bytes,         ;
;                         Low byte of subscript (dim N-1),              ;
;                         High byte of subscript (dim N-1),             ;
```

```
;                                                                       ;
;                                  .                                    ;
;                                  .                                    ;
;                                  .                                    ;
;                       Low byte of size (dim 0) in bytes,              ;
;                       High byte of size (dim 0) in bytes,             ;
;                       Low byte of subscript (dim 0),                  ;
;                       High byte of subscript (dim 0),                 ;
;                       Low byte of base address of array,              ;
;                       High byte of base address of array              ;
;                                                                       ;
;       Exit:           Register A = High byte of address               ;
;                       Register Y = Low byte of address                ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Approximately 1100 cycles per dimension         ;
;                       plus 90 cycles overhead.                        ;
;                                                                       ;
;       Size:           Program 192 bytes                               ;
;                       Data     11 bytes                               ;
;                                                                       ;
;                                                                       ;

NDIM:
        ;POP PARAMETERS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1        ;SAVE RETURN ADDRESS

        PLA
        STA     NUMDIM          ;GET NUMBER OF DIMENSIONS

        ;OFFSET := 0
        LDA     #0
        STA     OFFSET
        STA     OFFSET+1

        ;CHECK FOR ZERO DIMENSIONS JUST IN CASE
        LDA     NUMDIM
        BEQ     ADBASE          ;ASSUME THERE IS A BASE ADDRESS EVEN
                                ; IF THERE ARE NO DIMENSIONS

        ;LOOP ON EACH DIMENSION
        ; DOING OFFSET := OFFSET + (SUBSCRIPT * SIZE)
LOOP:
        PLA                     ;POP SIZE
        STA     SIZE
        PLA
        STA     SIZE+1

        PLA                     ;POP SUBSCRIPT
        STA     SS
        PLA
```

```
        STA     SS+1

        JSR     NXTOFF          ;OFFSET := OFFSET + (SUBSCRIPT * SIZE)

        DEC     NUMDIM          ;DECREMENT NUMBER OF DIMENSIONS
        BNE     LOOP            ;CONTINUE THROUGH ALL DIMENSIONS

ADBASE:
        ;CALCULATE THE STARTING ADDRESS OF THE ELEMENT
        ;OFFSET = BASE + OFFSET
        PLA                     ;GET LOW BYTE OF BASE
        CLC
        ADC     OFFSET          ;ADD LOW BYTE OF OFFSET
        STA     OFFSET
        PLA                     ;GET HIGH BYTE OF BASE
        ADC     OFFSET+1        ;A = HIGH BYTE OF BASE + OFFSET
        STA     OFFSET+1

        ;RESTORE RETURN ADDRESS AND EXIT
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA
        LDA     OFFSET+1        ;RETURN THE ADDRESS WHICH IS IN OFFSET
        LDY     OFFSET
        RTS

;
;
;SUBROUTINE NXTOFF
;PURPOSE: OFFSET := OFFSET + (SUBSCRIPT * SIZE);
;ENTRY: OFFSET = CURRENT OFFSET
;       SUBSCRIPT = CURRENT SUBSCRIPT
;       SIZE = CURRENT SIZE OF THIS DIMENSION
;EXIT:  OFFSET = OFFSET + (SUBSCRIPT * SIZE);
;REGISTERS USED: ALL
;
NXTOFF:
        ;
        ;CHECK IF SIZE IS POWER OF 2 OR 8 (EASY MULTIPLICATIONS - SHIFT ONLY)
        LDA     SIZE+1          ;HIGH BYTE = 0 ?
        BNE     BIGSZ           ;BRANCH IF SIZE IS LARGE

        LDA     SIZE
        LDY     #0              ;Y=INDEX INTO EASY ARRAY
        LDX     #SZEASY         ;X=SIZE OF EASY ARRAY
EASYLP:
        CMP     EASYAY,Y
        BEQ     ISEASY          ;BRANCH IF SIZE IS AN EASY ELEMENT
        INY                     ;INCREMENT INDEX
        DEX                     ;DECREMENT COUNT
        BNE     EASYLP          ;BRANCH IF NOT THROUGH ALL EASY ELEMENTS
        BEQ     BIGSZ           ;BRANCH IF SIZE IS NOT EASY
```

```
ISEASY:
        CPY     #0
        BEQ     ADDOFF          ;BRANCH IF SHIFT FACTOR = 0

        ;ELEMENT SIZE * SUBSCRIPT CAN BE PERFORMED WITH A SHIFT LEFT
SHL:
        ASL     SS              ;SHIFT LEFT LOW BYTE
        ROL     SS+1            ;SHIFT LEFT HIGH BYTE
        DEY
        BNE     SHL             ;CONTINUE UNTIL DONE
        BEQ     ADDOFF          ;DONE SO ADD OFFSET + SUBSCRIPT


BIGSZ:
        ;SIZE IS NOT AN EASY MULTIPLICATION SO PERFORM MULTIPLICATION OF
        ; ELEMENT SIZE AND SUBSCRIPT THE HARD WAY
        LDA     #0              ;PARTIAL PRODUCT = ZERO INITIALLY
        STA     PROD
        STA     PROD+1
        LDX     #17             ;NUMBER OF SHIFTS = 17
        CLC
MULLP:
        ROR     PROD+1          ;SHIFT PARTIAL PRODUCT
        ROR     PROD
        ROR     SS+1            ;SHIFT MULTIPLIER
        ROR     SS
        BCC     DECCNT
        CLC                     ;ADD MULTIPLICAND TO PARTIAL PRODUCT
        LDA     SIZE            ; IF NEXT BIT OF MULTIPLIER IS 1
        ADC     PROD
        STA     PROD
        LDA     SIZE+1
        ADC     PROD+1
        STA     PROD+1

DECCNT:
        DEX
        BNE     MULLP


ADDOFF:
        LDA     SS
        CLC
        ADC     OFFSET          ;ADD LOW BYTES
        STA     OFFSET
        LDA     SS+1
        ADC     OFFSET+1        ;ADD HIGH BYTES
        STA     OFFSET+1

        RTS

EASYAY:                 ;SHIFT FACTOR
        .BYTE   1       ;0
        .BYTE   2       ;1
        .BYTE   4       ;2
```

```
                .BYTE   8       ;3
                .BYTE   16.     ;4
                .BYTE   32.     ;5
                .BYTE   64.     ;6
                .BYTE   128.    ;7
SZEASY          .EQU    $-EASYAY

;
;DATA
RETADR:         .BLOCK  2       ;TEMPORARY FOR RETURN ADDRESS
SS:             .BLOCK  2       ;SUBSCRIPT INTO THE ARRAY
SIZE:           .BLOCK  2       ;SIZE OF AN ARRAY ELEMENT
OFFSET:         .BLOCK  2       ;TEMPORARY FOR CALCULATING
NUMDIM:         .BLOCK  1       ;NUMBER OF DIMENSIONS
PROD:           .BLOCK  2       ;TEMPORARY FOR MULTIPLICATION IN NXTOFF

;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

;PROGRAM SECTION
SC0507:
;
;FIND ADDRESS OF AY1[1,3,0]
;  SINCE LOWER BOUNDS OF ARRAY 1 ARE ALL ZERO IT IS NOT
;  NECESSARY TO NORMALIZE THEM

;PUSH BASE ADDRESS OF ARRAY 1
LDA     AY1ADR+1
PHA
LDA     AY1ADR
PHA

;PUSH SUBSCRIPT AND SIZE FOR DIMENSION 1
LDA     #0
PHA
LDA     #1
PHA
LDA     #0
PHA
LDA     #A1SZ1
PHA

;PUSH SUBSCRIPT AND SIZE FOR DIMENSION 2
LDA     #0
PHA
LDA     #3
PHA
LDA     #0
PHA
LDA     #A1SZ2
PHA
```

```
;PUSH SUBSCRIPT AND SIZE FOR DIMENSION 3
LDA     #0
PHA
LDA     #0
PHA
LDA     #0
PHA
LDA     #A1SZ3
PHA

;PUSH NUMBER OF DIMENSIONS
LDA     #A1DIM
PHA

JSR     NDIM            ;CALCULATE ADDRESS
BRK                     ;AY = STARTING ADDRESS OF ARY1(1,3,0)
                        ;   = ARY + (1*126) + (3*21) + (0*3)
                        ;   = ARY + 189

;
;CALCULATE ADDRESS OF AY2[-1,6]
;  SINCE LOWER BOUNDS OF AY 2 DO NOT START AT ZERO THE SUBSCRIPTS
;  MUST BE NORMALIZED
;PUSH BASE ADDRESS OF ARRAY 2
LDA     AY2ADR+1
PHA
LDA     AY2ADR
PHA

;PUSH (SUBSCRIPT - LOWER BOUND) AND SIZE FOR DIMENSION 1
LDA     #-1
SEC
SBC     #A2D1L
TAX                     ;SAVE LOW BYTE
LDA     #0FFH           ;HIGH BYTE OF -1 SUBSCRIPT
SBC     #0FFH           ;HIGH BYTE OF A2D1L
PHA                     ;PUSH HIGH BYTE
TXA
PHA                     ;PUSH LOW BYTE
LDA     #0
PHA
LDA     #A2SZ1
PHA

;PUSH (SUBSCRIPT - LOWER BOUND) AND SIZE FOR DIMENSION 2
LDA     #6
SEC
SBC     #A2D2L
TAX                     ;SAVE LOW BYTE
LDA     #0
SBC     #0
PHA                     ;PUSH HIGH BYTE
TXA
PHA                     ;PUSH LOW BYTE
LDA     #0
```

```
        PHA
        LDA     #A2SZ2
        PHA

        ;PUSH NUMBER OF DIMENSIONS
        LDA     #A2DIM
        PHA

        JSR     NDIM    ;CALCULATE ADDRESS
        BRK             ;AY = STARTING ADDRESS OF ARY1(-1,6)
                        ;   = ARY + (((-1) - (-5))*18) + ((6 - 2)*2)
                        ;   = ARY + 80

        JMP     SC0507

;DATA
AY1ADR: .WORD   AY1     ;ADDRESS OF ARRAY 1
AY2ADR: .WORD   AY2     ;ADDRESS OF ARRAY 2

;
;AY1 : ARRAY[A1D1L..A1D1H,A1D2L..A1D2H,A1D3L..A1D3H] OF THREE BYTE ELEMENTS
;           [  0  ..  3  ,  0  ..  5  ,  0  ..  6  ]
A1DIM:  .EQU    3                       ;NUMBER OF DIMENSIONS OF ARRAY 1
A1D1L:  .EQU    0                       ;LOW BOUND OF ARRAY 1 DIMENSION 1
A1D1H:  .EQU    3                       ;HIGH BOUND OF ARRAY 1 DIMENSION 1
A1D2L:  .EQU    0                       ;LOW BOUND OF ARRAY 1 DIMENSION 2
A1D2H:  .EQU    5                       ;HIGH BOUND OF ARRAY 1 DIMENSION 2
A1D3L:  .EQU    0                       ;LOW BOUND OF ARRAY 1 DIMENSION 3
A1D3H:  .EQU    6                       ;HIGH BOUND OF ARRAY 1 DIMENSION 3
A1SZ3:  .EQU    3                       ;SIZE OF AN ELEMENT IN DIMENSION 3
A1SZ2:  .EQU    ((A1D3H-A1D3L)+1)*A1SZ3 ;SIZE OF AN ELEMENT IN DIMENSION 2
A1SZ1:  .EQU    ((A1D2H-A1D2L)+1)*A1SZ2 ;SIZE OF AN ELEMENT IN DIMENSION 1
AY1:    .BLOCK  ((A1D1H-A1D1L)+1)*A1SZ1 ;THE ARRAY

;AY2 : ARRAY[A1D1L..A1D1H,A1D2L..A1D2H] OF WORD
;           [ -5 ..  -1  ,  2  .. 10 ]
A2DIM:  .EQU    2                       ;NUMBER OF DIMENSIONS OF ARRAY 2
A2D1L:  .EQU    -5                      ;LOW BOUND OF ARRAY 2 DIMENSION 1
A2D1H:  .EQU    -1                      ;HIGH BOUND OF ARRAY 2 DIMENSION 1
A2D2L:  .EQU    2                       ;LOW BOUND OF ARRAY 2 DIMENSION 2
A2D2H:  .EQU    10                      ;HIGH BOUND OF ARRAY 2 DIMENSION 2
A2SZ2:  .EQU    2                       ;SIZE OF AN ELEMENT IN DIMENSION 2
A2SZ1:  .EQU    ((A2D2H-A2D2L)+1)*A2SZ2 ;SIZE OF AN ELEMENT IN DIMENSION 1
AY2:    .BLOCK  ((A2D1H-A2D1L)+1)*A2SZ1 ;THE ARRAY

        .END    ;PROGRAM
```

16-Bit Addition (ADD16)

Adds two 16-bit operands obtained from the stack and places the sum at the top of the stack. All 16-bit numbers are stored in the usual 6502 style with the less significant byte on top of the more significant byte.

Procedure: The program clears the Carry flag initially and adds the operands one byte at a time, starting with the less significant bytes. It sets the Carry flag from the addition of the more significant bytes.

Registers Used: A, P, Y

Execution Time: 80 cycles

Program Size: 38 bytes

Data Memory Required: Four bytes anywhere in memory for the second operand (two bytes starting at address ADEND2) and the return address (two bytes starting at address RETADR).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of first operand
More significant byte of first operand

Less significant byte of second operand
More significant byte of second operand

Exit Conditions

Order in stack (starting from the top)

Less significant byte of sum
More significant byte of sum

Examples

1. Data: First operand = $03E1_{16}$
   Second operand = $07E4_{16}$

   Result: Sum = $0BC5_{16}$
   Carry = 0

2. Data: First operand = $A45D_{16}$
   Second operand = $97E1_{16}$

   Result: Sum = $3C3E_{16}$
   Carry = 1

```
;       Title           16 bit addition                                 ;
;       Name:           ADD16                                           ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Add 2 16 bit signed or unsigned words and return;
;                       a 16 bit signed or unsigned sum.                ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of operand 2,                        ;
;                         High byte of operand 2,                       ;
;                         Low byte of operand 1,                        ;
;                         High byte of operand 1                        ;
;                                                                       ;
;       Exit:           Sum = operand 1 + operand 2                     ;
;                       TOP OF STACK                                    ;
;                         Low byte of sum,                              ;
;                         High byte of sum                              ;
;                                                                       ;
;       Registers used: A,P,Y                                           ;
;                                                                       ;
;       Time:           80 cycles                                       ;
;                                                                       ;
;       Size:           Program 38 bytes                                ;
;                       Data     4 bytes                                ;
;                                                                       ;
;                                                                       ;

ADD16:
        ;SAVE THE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET ADDEND 2
        PLA
        STA     ADEND2
        PLA
        STA     ADEND2+1

        ;SUM ADDEND 2 WITH ADDEND 1
        PLA
        CLC
        ADC     ADEND2
        TAY                     ;SAVE LOW BYTE OF SUM
        PLA
        ADC     ADEND2+1

        ;PUSH THE SUM
        PHA                     ;PUSH HIGH BYTE
        TYA
```

```
        PHA                         ;PUSH LOW BYTE

        ;PUSH RETURN ADDRESS AND EXIT
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA
        RTS

;DATA
ADEND2: .BLOCK  2                   ;TEMPORARY FOR ADDEND 2
RETADR: .BLOCK  2                   ;TEMPORARY FOR RETURN ADDRESS

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;


SC0601:
        ;SUM OPRND1 + OPRND2
        LDA     OPRND1+1
        PHA
        LDA     OPRND1
        PHA
        LDA     OPRND2+1
        PHA
        LDA     OPRND2
        PHA
        JSR     ADD16
        PLA
        TAY
        PLA
        BRK                         ;A = HIGH BYTE, Y = LOW BYTE
        JMP     SC0601

;TEST DATA, CHANGE FOR DIFFERENT VALUES
OPRND1  .WORD   1023                ;1023 + 123 = 1146 = 047AH
OPRND2  .WORD   123

        .END    ;PROGRAM
```

16-Bit Subtraction (SUB16) 6B

Subtracts two 16-bit operands obtained from the stack and places the difference at the top of the stack. All 16-bit numbers are stored in the usual 6502 style with the less significant byte on top of the more significant byte. The subtrahend (number to be subtracted) is stored on top of the minuend (number from which the subtrahend is subtracted). The Carry flag acts as an inverted borrow, its usual role in the 6502.

Procedure: The program sets the Carry flag (the inverted borrow) initially and subtracts the subtrahend from the minuend one byte at a time, starting with the less significant bytes. It sets the Carry flag from the subtraction of the more significant bytes.

Registers Used: A, P, Y

Execution Time: 80 cycles

Program Size: 38 bytes

Data Memory Required: Four bytes anywhere in memory for the subtrahend (two bytes starting at address SUBTRA) and the return address (two bytes starting at address RETADR).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of subtrahend
More significant byte of subtrahend

Less significant byte of minuend
More significant byte of minuend

Exit Conditions

Order in stack (starting from the top)

Less significant byte of difference (minuend − subtrahend)
More significant byte of difference (minuend − subtrahend)

Examples

1. Data: Minuend = $A45D_{16}$
   Subtrahend = $97E1_{16}$

   Result: Difference = Minuend − Subtrahend
   = $0C7C_{16}$
   Carry = 1 (no borrow)

2. Data: Minuend = $03E1_{16}$
   Subtrahend = $07E4_{16}$

   Result: Difference = Minuend − Subtrahend
   = $FBFD_{16}$
   Carry = 0 (borrow generated)

```
;       Title           16 bit subtraction                                  ;
;       Name:           SUB16                                               ;
;                                                                           ;
;                                                                           ;

;                                                                           ;
;       Purpose:        Subtract 2 16 bit signed or unsigned words and      ;
;                       return a 16 bit signed or unsigned difference.      ;
;                                                                           ;
;       Entry:          TOP OF STACK                                        ;
;                         Low byte of return address,                       ;
;                         High byte of return address,                      ;
;                         Low byte of subtrahend,                           ;
;                         High byte of subtrahend,                          ;
;                         Low byte of minuend,                              ;
;                         High byte of minuend                              ;
;                                                                           ;
;       Exit:           Difference = minuend - subtrahend                   ;
;                       TOP OF STACK                                        ;
;                         Low byte of difference,                           ;
;                         High byte of difference                           ;
;                                                                           ;
;       Registers used: A,P,Y                                               ;
;                                                                           ;
;       Time:           80 cycles                                           ;
;                                                                           ;
;       Size:           Program 38 bytes                                    ;
;                       Data     4 bytes                                    ;
;                                                                           ;
;                                                                           ;

SUB16:
        ;SAVE THE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET SUBTRAHEND
        PLA
        STA     SUBTRA
        PLA
        STA     SUBTRA+1

        ;SUBTRACT SUBTRAHEND FROM MINUEND
        PLA
        SEC
        SBC     SUBTRA
        TAY                     ;SAVE LOW BYTE OF THE DIFFERENCE
        PLA
        SBC     SUBTRA+1

        ;PUSH THE DIFFERENCE
        PHA                     ;PUSH HIGH BYTE
        TYA
        PHA                     ;PUSH LOW BYTE
```

```
        ;PUSH RETURN ADDRESS AND EXIT
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA
        RTS

;DATA
SUBTRA: .BLOCK  2               ;TEMPORARY FOR SUBTRAHEND
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;

SC0602:
        ;SUBTRACT OPRND2 FROM OPRND1
        LDA     OPRND1+1
        PHA
        LDA     OPRND1
        PHA
        LDA     OPRND2+1
        PHA
        LDA     OPRND2
        PHA
        JSR     SUB16
        PLA
        TAY
        PLA
        BRK                     ;A = HIGH BYTE, Y = LOW BYTE
        JMP     SC0602

;TEST DATA - CHANGE TO TEST OTHER VALUES
OPRND1  .WORD   123             ;123 - 1023 = -900 = 0FC7CH
OPRND2  .WORD   1023

        .END    ;PROGRAM
```

16-Bit Multiplication (MUL16) 6C

Multiplies two 16-bit operands obtained from the stack and places the less significant word of the product at the top of the stack. All 16-bit numbers are stored in the usual 6502 style with the less significant byte on top of the more significant byte.

Procedure: The program uses an ordinary add-and-shift algorithm, adding the multiplicand to the partial product each time it finds a 1 bit in the multiplier. The partial product and the multiplier are shifted 17 times (the number of bits in the multiplier plus 1) with the extra loop being necessary to move the final Carry into the product. The program maintains a full 32-bit unsigned partial product in memory locations (starting with the most significant byte) HIPROD+1, HIPROD, MLIER+1, and MLIER. The less significant word of the product replaces the multiplier as the multiplier is shifted and examined for 1 bits.

Registers Used: All

Execution Time: Approximately 650 to 1100 cycles, depending largely on the number of 1 bits in the multiplier.

Program Size: 238 bytes

Data Memory Required: Eight bytes anywhere in memory for the multiplicand (two bytes starting at address MCAND), the multiplier and less significant word of the partial product (two bytes starting at address MLIER), the more significant word of the partial product (two bytes starting at address HIPROD), and the return address (two bytes starting at address RETADR).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of multiplier
More significant byte of multiplier

Less significant byte of multiplicand
More significant byte of multiplicand

Exit Conditions

Order in stack (starting from the top)

Less significant byte of less significant word of product
More significant byte of less significant word of product

Examples

1. Data: Multiplier = 0012_{16} (18_{10})
   Multiplicand = $03D1_{16}$ (977_{10})

   Result: Product = $44B2_{16}$ ($17,586_{10}$)

2. Data: Multiplier = $37D1_{16}$ ($14,289_{10}$)
   Multiplicand = $A045_{16}$ ($41,029_{10}$)

   Result: Product = $AB55_{16}$ ($43,861_{10}$). This is actually the less significant 16-bit word of the 32-bit product $22F1AB55_{16}$ ($586,264,381_{10}$).

Note that MUL16 returns only the less significant word of the product to maintain compatibility with other 16-bit arithmetic operations. The more significant word of the product is available in memory locations HIPROD (less significant byte) and HIPROD+1 (more significant byte), but the user should note that it is correct only if the operands are unsigned. If the operands are signed numbers and either one is negative, the user must determine the sign of the product and replace negative operands with their absolute values (two's complements) before calling MUL16.

```
;       Title           16 bit Multiplication                          ;
;       Name:           MUL16                                          ;
;                                                                      ;
;                                                                      ;
;
;       Purpose:        Multiply 2 signed or unsigned 16 bit words and ;
;                       return a 16 bit signed or unsigned product.    ;
;       Entry:          TOP OF STACK                                   ;
;                         Low byte of return address,                  ;
;                         High byte of return address,                 ;
;                         Low byte of multiplier,                      ;
;                         High byte of multiplier,                     ;
;                         Low byte of multiplicand,                    ;
;                         High byte of multiplicand                    ;
;       Exit:           Product = multiplicand * multiplier            ;
;                       TOP OF STACK                                   ;
;                         Low byte of product,                         ;
;                         High byte of product,                        ;
;       Registers used: All                                            ;
;       Time:           Approximately 650 to 1100 cycles               ;
;       Size:           Program 238 bytes                              ;
;                       Data      8 bytes                              ;
;                                                                      ;
;

MUL16:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET MULTIPLIER
        PLA
        STA     MLIER
        PLA
```

```
        STA     MLIER+1

        ;GET MULTIPLICAND
        PLA
        STA     MCAND
        PLA
        STA     MCAND+1
        ;PERFORM MULTIPLICATION USING THE SHIFT AND ADD ALGORITHM
        ; THIS ALGORITHM PRODUCES A UNSIGNED 32 BIT PRODUCT IN
        ; HIPROD AND MLIER WITH HIPROD BEING THE HIGH WORD.
        LDA     #0
        STA     HIPROD          ;ZERO HIGH WORD OF PRODUCT
        STA     HIPROD+1
        LDX     #17             ;NUMBER OF BITS IN MULTIPLIER PLUS 1, THE
                                ; EXTRA LOOP IS TO MOVE THE LAST CARRY INTO
                                ; THE PRODUCT
        CLC                     ; CLEAR CARRY FOR FIRST TIME THROUGH LOOP
MULLP:
        ;
        ;IF NEXT BIT = 1 THEN
        ;  HIPROD := HIPROD + MULTIPLICAND
        ROR     HIPROD+1
        ROR     HIPROD
        ROR     MLIER+1
        ROR     MLIER
        BCC     DECCNT          ;BRANCH IF NEXT BIT OF MULTIPLIER IS 0

        CLC                     ;NEXT BIT IS 1 SO ADD MULTIPLICAND TO PRODUCT
        LDA     MCAND
        ADC     HIPROD
        STA     HIPROD
        LDA     MCAND+1
        ADC     HIPROD+1
        STA     HIPROD+1        ;CARRY = OVERFLOW FROM ADD

DECCNT:
        DEX
        BNE     MULLP           ;CONTINUE UNTIL DONE


        ;PUSH LOW WORD OF PRODUCT ON TO STACK
        LDA     MLIER+1
        PHA
        LDA     MLIER
        PHA

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA
        RTS


;DATA
MCAND:  .BLOCK  2               ;MULTIPLICAND
```

```
MLIER:   .BLOCK   2              ;MULTIPLIER AND LOW WORD OF PRODUCT
HIPROD:  .BLOCK   2              ;HIGH WORD OF PRODUCT
RETADR:  .BLOCK   2              ;RETURN ADDRESS
;                                                                    ;
;                                                                    ;
;        SAMPLE EXECUTION:                                           ;
;                                                                    ;
;                                                                    ;

SC0603:
         ;MULTIPLY OPRND1 * OPRND2 AND STORE THE PRODUCT AT RESULT
         LDA     OPRND1+1
         PHA
         LDA     OPRND1
         PHA
         LDA     OPRND2+1
         PHA
         LDA     OPRND2
         PHA
         JSR     MUL16           ;MULTIPLY
         PLA
         STA     RESULT
         PLA
         STA     RESULT+1
         BRK                     ;RESULT OF 1023 * -2 = -2046 = 0F802H
                                 ; IN MEMORY RESULT   = 02H
                                 ;           RESULT+1 = F8H
         JMP     SC0603

OPRND1   .WORD   -2
OPRND2   .WORD   1023
RESULT:  .BLOCK  2               ;2 BYTE RESULT

         .END    ;PROGRAM
```

16-Bit Division
(SDIV16, UDIV16, SREM16, UREM16) 6D

Divides two 16-bit operands obtained from the stack and places either the quotient or the remainder at the top of the stack. There are four entry points: SDIV16 returns a 16-bit signed quotient from dividing two 16-bit signed operands, UDIV16 returns a 16-bit unsigned quotient from dividing two 16-bit unsigned operands, SREM16 returns a 16-bit remainder (a signed number) from dividing two 16-bit signed operands, and UREM16 returns a 16-bit unsigned remainder from dividing two 16-bit unsigned operands. All 16-bit numbers are stored in the usual 6502 style with the less significant byte on top of the more significant byte. The divisor is stored on top of the dividend. If the divisor is zero, the Carry flag is set and a zero result is returned; otherwise, the Carry flag is cleared.

Ragisters Used: All
Execution Tima: Approximately 1000 to 1160 cycles, depending largely on the number of trial subtractions that are successful and thus require the replacement of the previous dividend by the remainder.
Program Size: 293 bytes
Data Memory Required: Eleven bytes anywhere in memory. These are utilized as follows: two bytes for the divisor (starting at address DVSOR); four bytes for the extended dividend (starting at address DVEND) and also for the quotient and remainder; two bytes for the return address (starting at address RETADR); one byte for the sign of the quotient (address SQUOT); one byte for the sign of the remainder (address SREM); and one byte for an index to the result (address RSLTIX).
Special Casa: If the divisor is zero, the program returns with the Carry flag set to 1 and a result of zero. Both the quotient and the remainder are zero.

Procedure: If the operands are signed, the program determines the sign of the quotient and takes the absolute values of any negative operands. It also must retain the sign of the dividend, since that determines the sign of the remainder. The program then performs the actual unsigned division by the usual shift-and-subtract algorithm, shifting quotient and dividend and placing a 1 bit in the quotient each time a trial subtraction is successful. If the operands are signed, the program must negate (that is, subtract from zero) any result (quotient or remainder) that is negative. The Carry flag is cleared if the division is proper and set if the divisor is found to be zero. A zero divisor also results in a return with the result (quotient or remainder) set to zero.

Entry Conditions
Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of divisor
More significant byte of divisor

Less significant byte of dividend
More significant byte of dividend

Exit Conditions
Order in stack (starting from the top)

Less significant byte of result
More significant byte of result

If the divisor is non-zero, Carry = 0 and the result is normal. If the divisor is zero, Carry = 1 and the result is 0000_{16}.

Examples

1. Data: Dividend = $03E0_{16} = 992_{10}$
   Divisor = $00B6_{16} = 182_{10}$

   Result: Quotient (from UDIV16) = 0005_{16}
   Remainder (from UREM16) = $0052_{16} = 0082_{10}$
   Carry = 0 (no divide-by-zero error)

2. Data: Dividend = $D73A_{16} = -10{,}438_{10}$
   Divisor = $02F1_{16} = 753_{10}$

   Result: Quotient (from SDIV16) = $FFF3_{16} = -13_{10}$
   Remainder (from SREM16) = $FD77_{16} = -649_{10}$
   Carry = 0 (no divide-by-zero error)

Note that we have taken the view that the remainder of a signed division may be either positive or negative. In our procedure, the remainder always takes the sign of the dividend. The user can easily examine the quotient and change the form to obtain a remainder that is always positive. In that case, the final result of Example 2 would be

Quotient = $FFF2_{16} = -14_{10}$
Remainder (always positive) = $0068_{16} = 104_{10}$

Regardless of the entry point used, the program always calculates both the quotient and the remainder. Upon return, the quotient is available in addresses DVEND and DVEND+1 (more significant byte in DVEND+1) and the remainder in addresses DVEND+2 and DVEND+3 (more significant byte in DVEND+3). Thus, the user can always obtain the result that is not returned in the stack.

```
;       Title           16 bit division                                 ;
;       Name:           SDIV16, UDIV16, SREM16, UREM16                  ;
;                                                                       ;
;                                                                       ;
;
;       Purpose:        SDIV16                                          ;
;                         Divide 2 signed 16 bit words and return a     ;
;                         16 bit signed quotient.                       ;
;                                                                       ;
;                       UDIV16                                          ;
;                         Divide 2 unsigned 16 bit words and return a   ;
;                         16 bit unsigned quotient.                     ;
;                                                                       ;
;                       SREM16                                          ;
;                         Divide 2 signed 16 bit words and return a     ;
;                         16 bit signed remainder.                      ;
;                                                                       ;
;                       UREM16                                          ;
;                         Divide 2 unsigned 16 bit words and return a   ;
;                         16 bit unsigned remainder.                    ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
```

```
;                         Low byte of return address,                    ;
;                         High byte of return address,                   ;
;                         Low byte of divisor,                           ;
;                         High byte of divisor,                          ;
;                         Low byte of dividend,                          ;
;                         High byte of dividend                          ;
;                                                                        ;
;       Exit:             TOP OF STACK                                   ;
;                         Low byte of result,                            ;
;                         High byte of result,                           ;
;                                                                        ;
;                         If no errors then                              ;
;                           carry := 0                                   ;
;                         else                                           ;
;                           divide by zero error                         ;
;                           carry := 1                                   ;
;                           quotient := 0                                ;
;                           remainder := 0                               ;
;                                                                        ;
;       Registers used:   All                                          ;
;                                                                        ;
;       Time:             Approximately 1000 to 1160 cycles            ;
;                                                                        ;
;       Size:             Program 293 bytes                            ;
;                         Data     13 bytes                            ;
;                                                                        ;
;                                                                        ;
;UNSIGNED DIVISION
UDIV16:
        LDA     #0              ;RESULT IS QUOTIENT (INDEX=0)
        BEQ     UDIVMD

;UNSIGNED REMAINDER
UREM16:
        LDA     #2              ;RESULT IS REMAINDER (INDEX=2)

UDIVMD:
        STA     RSLTIX          ;RESULT INDEX (0 FOR QUOTIENT,
                                ;                2 FOR REMAINDER)

        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET DIVISOR
        PLA
        STA     DVSOR
        PLA
        STA     DVSOR+1

        ;GET DIVIDEND
        PLA
        STA     DVEND
        PLA
```

```
        STA     DVEND+1

        ;PERFORM DIVISION
        JSR     UDIV
        BCC     DIVOK           ;BRANCH IF NO ERRORS
DIVER:  JMP     EREXIT
DIVOK:  JMP     OKEXIT

;
;SIGNED DIVISION
SDIV16:
        LDA     #0              ;RESULT IS QUOTIENT (INDEX=0)
        BEQ     SDIVMD

;SIGNED REMAINDER
SREM16:
        LDA     #2              ;RESULT IS REMAINDER (INDEX=2)
        BNE     SDIVMD

SDIVMD:
        STA     RSLTIX          ;RESULT INDEX (0 FOR QUOTIENT,
                                ;                  2 FOR REMAINDER)

        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET DIVISOR
        PLA
        STA     DVSOR
        PLA
        STA     DVSOR+1

        ;GET DIVIDEND
        PLA
        STA     DVEND
        PLA
        STA     DVEND+1

        ;DETERMINE SIGN OF QUOTIENT BY PERFORMING AN EXCLUSIVE OR OF THE
        ; HIGH BYTES. IF THE SIGNS ARE THE SAME THEN BIT 7 WILL BE 0 AND THE
        ; QUOTIENT IS POSITIVE. IF THE SIGNS ARE DIFFERENT THEN THE QUOTIENT
        ; IS NEGATIVE.
        LDA     DVEND+1
        EOR     DVSOR+1
        STA     SQUOT

        ;SIGN OF REMAINDER IS THE SIGN OF THE DIVIDEND
        LDA     DVEND+1
        STA     SREM

        ;TAKE THE ABSOLUTE VALUE OF THE DIVISOR
        LDA     DVSOR+1
        BPL     CHKDE           ;BRANCH IF ALREADY POSITIVE
```

```
        LDA     #0              ;SUBTRACT DIVISOR FROM ZERO
        SEC
        SBC     DVSOR
        STA     DVSOR
        LDA     #0
        SBC     DVSOR+1
        STA     DVSOR+1

        ;TAKE THE ABSOLUTE VALUE OF THE DIVIDEND
CHKDE:
        LDA     DVEND+1
        BPL     DODIV           ;BRANCH IF DIVIDEND IS POSITIVE
        LDA     #0              ;SUBTRACT DIVIDEND FROM ZERO
        SEC
        SBC     DVEND
        STA     DVEND
        LDA     #0
        SBC     DVEND+1
        STA     DVEND+1

        ;DIVIDE ABSOLUTE VALUES
DODIV:  JSR     UDIV
        BCS     EREXIT          ;EXIT IF DIVIDE BY ZERO

        ;NEGATE QUOTIENT IF IT IS NEGATIVE
        LDA     SQUOT
        BPL     DOREM           ;BRANCH IF QUOTIENT IS POSITIVE
        LDA     #0              ;SUBTRACT QUOTIENT FROM ZERO
        SEC
        SBC     DVEND
        STA     DVEND
        LDA     #0
        SBC     DVEND+1
        STA     DVEND+1

DOREM:
        ;NEGATE REMAINDER IF IT IS NEGATIVE
        LDA     SREM
        BPL     OKEXIT          ;BRANCH IF REMAINDER IS POSITIVE
        LDA     #0
        SEC
        SBC     DVEND+2
        STA     DVEND+2
        LDA     #0
        SBC     DVEND+3
        STA     DVEND+3
        JMP     OKEXIT

        ;ERROR EXIT (CARRY = 1, RESULTS ARE ZERO)
EREXIT:
        LDA     #0
        STA     DVEND
        STA     DVEND+1         ;QUOTIENT := 0
        STA     DVEND+2
        STA     DVEND+3         ;REMAINDER := 0
        SEC                     ;CARRY = 1 IF ERROR
```

```
        BCS     DVEXIT

        ;GOOD EXIT (CARRY = 0)
OKEXIT:
        CLC                         ;CARRY = 0, NO ERRORS

DVEXIT:
        ;PUSH RESULT
        LDX     RSLTIX              ;GET INDEX TO RESULT (0=QUOTIENT, 2=REMAINDER)
        LDA     DVEND+1,X
        PHA
        LDA     DVEND,X
        PHA

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA
        RTS

;*************************************
;ROUTINE: UDIV
;PURPOSE: DIVIDE A 16 BIT DIVIDEND BY A 16 BIT DIVISOR
;ENTRY: DVEND = DIVIDEND
;       DVSOR = DIVISOR
;EXIT:  DVEND = QUOTIENT
;       DVEND+2 = REMAINDER
;REGISTERS USED: ALL
;*************************************

UDIV:
        ;ZERO UPPER WORD OF DIVIDEND THIS WILL BE CALLED DIVIDEND[1] BELOW
        LDA     #0
        STA     DVEND+2
        STA     DVEND+3

        ;FIRST CHECK FOR DIVISION BY ZERO
        LDA     DVSOR
        ORA     DVSOR+1
        BNE     OKUDIV              ;BRANCH IF DIVISOR IS NOT ZERO
        SEC                         ;ELSE ERROR EXIT
        RTS

        ;PERFORM THE DIVISION BY TRIAL SUBTRACTIONS
OKUDIV:
        LDX     #16                 ;LOOP THROUGH 16 BITS
DIVLP:
        ROL     DVEND               ;SHIFT THE CARRY INTO BIT 0 OF DIVIDEND
        ROL     DVEND+1             ;WHICH WILL BE THE QUOTIENT
        ROL     DVEND+2             ;AND SHIFT DIVIDEND AT THE SAME TIME
        ROL     DVEND+3

        ;
        ;CHECK IF DIVIDEND[1] IS LESS THAN DIVISOR
```

```
CHKLT:
        SEC
        LDA     DVEND+2
        SBC     DVSOR
        TAY                     ;SAVE LOW BYTE IN REG Y
        LDA     DVEND+3
        SBC     DVSOR+1         ;SUBTRACT HIGH BYTES WITH RESULT IN REG A
        BCC     DECCNT          ;BRANCH IF DIVIDEND < DIVISOR AND CARRY
        STY     DVEND+2         ;ELSE
        STA     DVEND+3         ;  DIVIDEND[1] := DIVIDEND[1] - DIVISOR

DECCNT:
        DEX
        BNE     DIVLP

        ROL     DVEND           ;SHIFT IN THE LAST CARRY FOR THE QUOTIENT
        ROL     DVEND+1
        CLC                     ;NO ERRORS, CLEAR CARRY
        RTS

;DATA
DVSOR:  .BLOCK  2               ;DIVISOR
DVEND:  .BLOCK  2               ;DIVIDEND[0] AND QUOTIENT
        .BLOCK  2               ;DIVIDEND[1] AND REMAINDER
RETADR: .BLOCK  2               ;RETURN ADDRESS
SQUOT:  .BLOCK  1               ;SIGN OF QUOTIENT
SREM:   .BLOCK  1               ;SIGN OF REMAINDER
RSLTIX: .BLOCK  1               ;INDEX TO THE RESULT 0 IS QUOTIENT,
                                ; 2 IS REMAINDER

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

;PROGRAM SECTION
SC0604:
        ;SIGNED DIVIDE, OPRND1 / OPRND2, STORE THE QUOTIENT AT QUOT
        LDA     OPRND1+1
        PHA
        LDA     OPRND1
        PHA
        LDA     OPRND2+1
        PHA
        LDA     OPRND2
        PHA
        JSR     SDIV16          ;SIGNED DIVIDE
        PLA
        STA     QUOT
        PLA
        STA     QUOT+1
        BRK                     ;RESULT OF -1023 / 123 = -8
                                ; IN MEMORY QUOT   = F8 HEX
                                ;           QUOT+1 = FF HEX
```

```
;UNSIGNED DIVIDE, OPRND1 / OPRND2, STORE THE QUOTIENT AT QUOT
LDA     OPRND1+1
PHA
LDA     OPRND1
PHA
LDA     OPRND2+1
PHA
LDA     OPRND2
PHA
JSR     UDIV16          ;UNSIGNED DIVIDE
PLA
STA     QUOT
PLA
STA     QUOT+1
BRK                     ;RESULT OF 64513 / 123 = 524
                        ; IN MEMORY QUOT   = 0C HEX
                        ;           QUOT+1 = 02 HEX

;SIGNED REMAINDER, OPRND1 / OPRND2, STORE THE REMAINDER AT REM
LDA     OPRND1+1
PHA
LDA     OPRND1
PHA
LDA     OPRND2+1
PHA
LDA     OPRND2
PHA
JSR     SREM16          ;REMAINDER
PLA
STA     REM
PLA
STA     REM+1
BRK                     ;THE REMAINDER OF -1023 / 123 = -39
                        ; IN MEMORY REM    = D9 HEX
                        ;           REM+1  = FF HEX

;UNSIGNED REMAINDER, OPRND1 / OPRND2, STORE THE REMAINDER AT REM
LDA     OPRND1+1
PHA
LDA     OPRND1
PHA
LDA     OPRND2+1
PHA
LDA     OPRND2
PHA
JSR     UREM16          ;REMAINDER
PLA
STA     REM
PLA
STA     REM+1
PLA
BRK                     ;THE REMAINDER OF 64513 / 123 = 61
                        ; IN MEMORY REM    = 3D HEX
                        ;           REM+1  = 00
```

```
        JMP     SC0604

;DATA
OPRND1  .WORD   -1023           ;DIVIDEND (64513 UNSIGNED)
OPRND2  .WORD   123             ;DIVISOR
QUOT:   .BLOCK  2               ;QUOTIENT
REM:    .BLOCK  2               ;REMAINDER

        .END    ;PROGRAM
```

16-Bit Comparison (CMP16) 6E

Compares two 16-bit operands obtained from the stack and sets the flags accordingly. All 16-bit numbers are stored in the usual 6502 style with the less significant byte on top of the more significant byte. The comparison is performed by subtracting the top operand (or subtrahend) from the bottom operand (or minuend). The Zero flag always indicates whether the numbers are equal. If the numbers are unsigned, the Carry flag indicates which one is larger (Carry = 0 if top operand or subtrahend is larger and 1 otherwise). If the numbers are signed, the Negative flag indicates which one is larger (Negative = 1 if top operand or subtrahend is larger and 0 otherwise); two's complement overflow is considered and the Negative flag is inverted if it occurs.

Procedure: The program first compares the less significant bytes of the subtrahend and the minuend. It then subtracts the more significant byte of the subtrahend from the more significant byte of the minuend, thus setting the flags. If the less significant bytes of the operands are not equal, the program clears the Zero flag by logically ORing the accumulator with 01_{16}. If the subtraction results in two's complement overflow, the program complements the Negative flag by logically Exclusive ORing the accumulator with 80_{16} (10000000_2); it also clears the Zero flag by the method described earlier.

Registers Used: A, P

Execution Time: Approximately 90 cycles

Program Size: 65 bytes

Data Memory Required: Six bytes anywhere in memory for the minuend or WORD1 (2 bytes starting at address MINEND), the subtrahend or WORD2 (2 bytes starting at address SUBTRA), and the return address (2 bytes starting at address RETADR).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of subtrahend (top operand or WORD2)
More significant byte of subtrahend (top operand or WORD2)

Less significant byte of minuend (bottom operand or WORD1)
More significant byte of minuend (bottom operand or WORD1)

Exit Conditions

Flags set as if subtrahend had been subtracted from minuend, with a correction if two's complement overflow occurred.

Zero flag = 1 if subtrahend and minuend are equal, 0 if they are not equal.

Carry flag = 0 if subtrahend is larger than minuend in the unsigned sense, 1 if it is less than or equal to the minuend.

Negative flag = 1 if subtrahend is larger than minuend in the signed sense, 0 if it is less than or equal to the minuend. This flag is corrected if two's complement overflow occurs.

Examples

1. Data: Minuend (bottom operand) = $03E1_{16}$
   Subtrahend (top operand) = $07E4_{16}$

   Result: Carry = 0, indicating subtrahend is larger in unsigned sense.
   Zero = 0, indicating operands not equal
   Negative = 1, indicating subtrahend is larger in signed sense

2. Data: Minuend (bottom operand) = $C51A_{16}$
   Subtrahend (top operand) = $C51A_{16}$

   Result: Carry = 1, indicating subtrahend is not larger in unsigned sense
   Zero = 1, indicating operands are equal
   Negative = 0, indicating subtrahend is not larger in signed sense

3. Data: Minuend (bottom operand) = $A45D_{16}$
   Subtrahend (top operand) = $77E1_{16}$

   Result: Carry = 1, indicating subtrahend is not larger in unsigned sense
   Zero = 0, indicating operands are not equal
   Negative = 1, indicating subtrahend is larger in signed sense

In Example 3, the bottom operand is a negative two's complement number, whereas the top operand is a positive two's complement number.

```
;       Title           16 bit compare                                  ;
;       Name:           CMP16                                           ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:        Compare 2 16 bit signed or unsigned words and   ;
;                       return the C,Z,N flags set or cleared.          ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of word 2 (subtrahend),              ;
;                         High byte of word 2 (subtrahend),             ;
;                         Low byte of word 1 (minuend),                 ;
;                         High byte of word 1 (minuend)                 ;
;                                                                       ;
;       Exit:           Flags returned based on word 1 - word 2         ;
;                       IF WORD1 AND WORD2 ARE 2'S COMPLEMENT NUMBERS   ;
;                       THEN                                            ;
;                        IF WORD1 = WORD2 THEN                          ;
;                          Z=1,N=0                                      ;
```

```
;                       IF WORD1 > WORD2 THEN                          ;
;                         Z=0,N=0                                      ;
;                       IF WORD1 < WORD2 THEN                          ;
;                         Z=0,N=1                                      ;
;                     ELSE                                             ;
;                       IF WORD1 = WORD2 THEN                          ;
;                         Z=1,C=1                                      ;
;                       IF WORD1 > WORD2 THEN                          ;
;                         Z=0,C=1                                      ;
;                       IF WORD1 < WORD2 THEN                          ;
;                         Z=0,C=0                                      ;
;                                                                      ;
;       Registers used: A,P                                            ;
;                                                                      ;
;       Time:           Approximately 90 cycles                        ;
;                                                                      ;
;       Size:           Program 65 bytes                               ;
;                       Data     6 bytes                               ;
;                                                                      ;
;                                                                      ;

CMP16:
        ;SAVE THE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET SUBTRAHEND
        PLA
        STA     SUBTRA
        PLA
        STA     SUBTRA+1

        ;GET MINUEND
        PLA
        STA     MINEND
        PLA
        STA     MINEND+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        LDA     MINEND
        CMP     SUBTRA          ;COMPARE LOW BYTES
        BEQ     EQUAL           ;BRANCH IF THEY ARE EQUAL

        ;LOW BYTES ARE NOT EQUAL - COMPARE HIGH BYTES
        LDA     MINEND+1
        SBC     SUBTRA+1        ;COMPARE HIGH BYTES
        ORA     #1              ;MAKE Z = 0, SINCE LOW BYTES ARE NOT EQUAL
        BVS     OVFLOW          ;MUST HANDLE OVERFLOW FOR SIGNED ARITHMATIC
        RTS                     ;EXIT
```

```
        ;LOW BYTES ARE EQUAL   - COMPARE HIGH BYTES
EQUAL:
        LDA     MINEND+1
        SBC     SUBTRA+1        ;UPPER BYTES
        BVS     OVFLOW          ;MUST HANDLE OVERFLOW FOR SIGNED ARITHMETIC
        RTS                     ;RETURN WITH FLAGS SET

        ;OVERFLOW WITH SIGNED ARITHMETIC SO COMPLEMENT THE NEGATIVE FLAG
        ; DO NOT CHANGE THE CARRY FLAG AND MAKE THE ZERO FLAG EQUAL 0.
        ; COMPLEMENT NEGATIVE FLAG BY EXCLUSIVE-ORING 80H AND ACCUMULATOR.
OVFLOW:
        EOR     #80H            ;COMPLEMENT NEGATIVE FLAG
        ORA     #1              ;IF OVERFLOW THEN THE WORDS ARE NOT EQUAL Z=0
                                ;CARRY UNCHANGED
        RTS



;DATA
MINEND: .BLOCK  2               ;TEMPORARY FOR THE MINUEND
SUBTRA: .BLOCK  2               ;TEMPORARY FOR THE SUBTRAHEND
RETADR: .BLOCK  2               ;TEMPORARY FOR THE RETURN ADDRESS

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;


SC0605:
        ;COMPARE OPRND1 AND OPRND2
        LDA     OPRND1+1
        PHA
        LDA     OPRND1
        PHA
        LDA     OPRND2+1
        PHA
        LDA     OPRND2
        PHA
        JSR     CMP16
        BRK                     ;LOOK AT THE FLAGS
                                ;  FOR 123 AND 1023
                                ;   C = 0, Z = 0, N = 1
        JMP     SC0605

OPRND1  .WORD   123             ;MINUEND
OPRND2  .WORD   1023            ;SUBTRAHEND

        .END    ;PROGRAM
```

Multiple-Precision Binary Addition (MPBADD) 6F

Adds two multi-byte unsigned binary numbers. Both numbers are stored with their least significant bytes first (at the lowest address). The sum replaces one of the numbers (the one with the starting address lower in the stack). The length of the numbers (in bytes) is 255 or less.

Procedure: The program clears the Carry flag initially and adds the operands one byte at a time, starting with the least significant bytes. The final Carry flag reflects the addition of the most significant bytes. The sum replaces the operand with the starting address lower in the stack (array 1 in the listing). A length of 00 causes an immediate exit with no addition operations.

Registers Used: All

Execution Time: 23 cycles per byte plus 82 cycles overhead. For example, adding two 6-byte operands takes 23 × 6 + 82 or 220 cycles

Program Size: 48 bytes

Data Memory Required: Two bytes anywhere in RAM plus four bytes on page 0. The two bytes anywhere in RAM are temporary storage for the return address (starting at address RETADR). The four bytes on page 0 hold pointers to the two numbers (starting at addresses AY1PTR and AY2PTR, respectively). In the listing, AY1PTR is taken as address $00D0_{16}$ and AY2PTR as address $00D2_{16}$.

Special Case: A length of zero causes an immediate exit with the sum equal to the bottom operand (i.e., array 1 is unchanged). The Carry flag is set to 1.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of second operand (address containing the least significant byte of array 2)
More significant byte of starting address of second operand (address containing the least significant byte of array 2)

Less significant byte of starting address of first operand and result (address containing the least significant byte of array 1)
More significant byte of starting address of first operand and result (address containing the least significant byte of array 1)

Exit Conditions

First operand (array 1) replaced by first operand (array 1) plus second operand (array 2).

Example

Data: Length of operands (in bytes) = 6
Top operand (array 2) = $19D028A193EA_{16}$
Bottom operand (array 1) = $293EABF059C7_{16}$

Result: Bottom operand (array 1) = Bottom operand (array 1) + Top operand (array 2) = $430ED491EDB1_{16}$
Carry = 0

```
;       Title           Multiple-Precision Binary Addition                  ;
;       Name:           MPBADD                                              ;
;                                                                           ;
;                                                                           ;
;                                                                           ;
;       Purpose:        Add 2 arrays of binary bytes                        ;
;                       Array1 := Array1 + Array2                           ;
;                                                                           ;
;       Entry:          TOP OF STACK                                        ;
;                         Low byte of return address,                       ;
;                         High byte of return address,                      ;
;                         Length of the arrays in bytes,                    ;
;                         Low byte of array 2 address,                      ;
;                         High byte of array 2 address,                     ;
;                         Low byte of array 1 address,                      ;
;                         High byte of array 1 address                      ;
;                                                                           ;
;                         The arrays are unsigned binary numbers with a     ;
;                         maximum length of 255 bytes, ARRAY[0] is the      ;
;                         least significant byte, and ARRAY[LENGTH-1]       ;
;                         the most significant byte.                        ;
;                                                                           ;
;       Exit:           Array1 := Array1 + Array2                           ;
;                                                                           ;
;       Registers used: All                                                 ;
;                                                                           ;
;       Time:           23 cycles per byte plus 82 cycles                   ;
;                       overhead.                                           ;
;                                                                           ;
;       Size:           Program 48 bytes                                    ;
;                       Data     2 bytes plus                               ;
;                                4 bytes in page zero                       ;
;                                                                           ;
;                                                                           ;

;EQUATES
AY1PTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 1 POINTER
AY2PTR: .EQU    0D2H            ;PAGE ZERO FOR ARRAY 2 POINTER

MPBADD:
        ;SAVE RETURN ADDRESS
```

```
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET LENGTH OF ARRAYS
        PLA
        TAX

        ;GET STARTING ADDRESS OF ARRAY 2
        PLA
        STA     AY2PTR
        PLA
        STA     AY2PTR+1

        ;GET STARTING ADDRESS OF ARRAY 1
        PLA
        STA     AY1PTR
        PLA
        STA     AY1PTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        LDY     #0
        CPX     #0              ;IS LENGTH OF ARRAYS = 0 ?
        BEQ     EXIT            ;YES, EXIT
        CLC                     ;CLEAR CARRY

LOOP:
        LDA     (AY1PTR),Y      ;GET NEXT BYTE
        ADC     (AY2PTR),Y      ;ADD BYTES
        STA     (AY1PTR),Y      ;STORE SUM
        INY                     ;INCREMENT ARRAY INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE UNTIL COUNTER = 0

EXIT:
        RTS

;
;DATA
RETADR  .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0606:
```

```
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH SIZE OF ARRAYS
        JSR     MPBADD          ;MULTIPLE-PRECISION BINARY ADDITION
        BRK                     ;RESULT OF 1234567H + 1234567H = 2468ACEH
                                ; IN MEMORY AY1     = CEH
                                ;           AY1+1   = 8AH
                                ;           AY1+2   = 46H
                                ;           AY1+3   = 02H
                                ;           AY1+4   = 00H
                                ;           AY1+5   = 00H
                                ;           AY1+6   = 00H
        JMP     SC0606

SZAYS:  .EQU    7       ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1     ;ADDRESS OF ARRAY 1
AY2ADR: .WORD   AY2     ;ADDRESS OF ARRAY 2

AY1:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Binary Subtraction (MPBSUB)

6G

Subtracts two multi-byte unsigned binary numbers. Both numbers are stored with their least significant byte at the lowest address. The starting address of the subtrahend (number to be subtracted) is stored on top of the starting address of the minuend (number from which the subtrahend is subtracted). The difference replaces the minuend in memory. The length of the numbers (in bytes) is 255 or less.

Procedure: The program sets the Carry flag (the inverted borrow) initially and subtracts the subtrahend from the minuend one byte at a time, starting with the least significant bytes. The final Carry flag reflects the subtraction of the most significant bytes. The difference replaces the minuend (the operand with the starting address lower in the stack, array 1 in the listing). A length of 00 causes an immediate exit with no subtraction operations.

Registers Used: All

Execution Time: 23 cycles per byte plus 82 cycles overhead. For example, subtracting two 6-byte operands takes $23 \times 6 + 82$ or 220 cycles.

Program Size: 48 bytes

Data Memory Required: Two bytes anywhere in RAM plus four bytes on page 0. The two bytes anywhere in RAM are temporary storage for the return address (starting at address RETADR). The four bytes on page 0 hold pointers to the two numbers (starting at addresses MINPTR and SUBPTR, respectively). In the listing, MINPTR is taken as address $00D0_{16}$ and SUBPTR as address $00D2_{16}$.

Special Case: A length of zero causes an immediate exit with the minuend unchanged (that is, the difference is equal to the bottom operand). The Carry flag is set to 1.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of subtrahend (address containing the least significant byte of array 2)
More significant byte of starting address of subtrahend (address containing the least significant byte of array 2)

Less significant byte of starting address of minuend (address containing the least significant byte of array 1)
More significant byte of starting address of minuend (address containing the least significant byte of array 1)

Exit Conditions

Minuend replaced by minuend minus subtrahend.

Example

Data: Length of operands (in bytes) = 4
Minuend = $2F5BA7C3_{16}$
Subtrahend = $14DF35B8_{16}$

Result: Difference = $1A7C720B_{16}$.
This number replaces the original minuend in memory. The Carry flag is set to 1 in accordance with its usual role (in 6502 programming) as an inverted borrow.

```
;       Title           Multiple-Precision Binary Subtraction                ;
;       Name:           MPBSUB                                               ;
;                                                                            ;
;                                                                            ;
;                                                                            ;
;       Purpose:        Subtract 2 arrays of binary bytes                    ;
;                       Minuend := Minuend - Subtrahend                      ;
;                                                                            ;
;       Entry:          TOP OF STACK                                         ;
;                         Low byte of return address,                        ;
;                         High byte of return address,                       ;
;                         Length of the arrays in bytes,                     ;
;                         Low byte of subtrahend address,                    ;
;                         High byte of subtrahend address,                   ;
;                         Low byte of minuend address,                       ;
;                         High byte of minuend address                       ;
;                                                                            ;
;                         The arrays are unsigned binary numbers with a      ;
;                         maximum length of 255 bytes, ARRAY[0] is the       ;
;                         least significant byte, and ARRAY[LENGTH-1]        ;
;                         the most significant byte.                         ;
;                                                                            ;
;       Exit:           Minuend := Minuend - Subtrahend                      ;
;                                                                            ;
;       Registers used: All                                                  ;
;                                                                            ;
;       Time:           23 cycles per byte plus 82 cycles                    ;
;                       overhead.                                            ;
;                                                                            ;
;       Size:           Program 48 bytes                                     ;
;                       Data     2 bytes plus                                ;
;                                4 bytes in page zero                        ;
;                                                                            ;
;                                                                            ;

;EQUATES
MINPTR: .EQU    0D0H            ;PAGE ZERO FOR MINUEND POINTER
SUBPTR: .EQU    0D2H            ;PAGE ZERO FOR SUBTRAHEND POINTER
```

```
MPBSUB:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET LENGTH OF ARRAYS
        PLA
        TAX

        ;GET STARTING ADDRESS OF SUBTRAHEND
        PLA
        STA     SUBPTR
        PLA
        STA     SUBPTR+1

        ;GET STARTING ADDRESS OF MINUEND
        PLA
        STA     MINPTR
        PLA
        STA     MINPTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        LDY     #0
        CPX     #0              ;IS LENGTH OF ARRAYS = 0 ?
        BEQ     EXIT            ;YES, EXIT
        SEC                     ;SET CARRY

LOOP:
        LDA     (MINPTR),Y      ;GET NEXT BYTE
        SBC     (SUBPTR),Y      ;SUBTRACT BYTES
        STA     (MINPTR),Y      ;STORE DIFFERENCE
        INY                     ;INCREMENT ARRAY INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE UNTIL COUNTER = 0

EXIT:
        RTS

;
;DATA
RETADR  .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS

;
;
;       SAMPLE EXECUTION:
;
;
```

```
SC0607:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH SIZE OF ARRAYS
        JSR     MPBSUB          ;MULTIPLE-PRECISION BINARY SUBTRACTION
        BRK                     ;RESULT OF 7654321H - 1234567H = 641FDBAH
                                ; IN MEMORY AY1     = 0BAH
                                ;           AY1+1   = 0FDH
                                ;           AY1+2   = 41H
                                ;           AY1+3   = 06H
                                ;           AY1+4   = 00H
                                ;           AY1+5   = 00H
                                ;           AY1+6   = 00H
        JMP     SC0607

SZAYS:  .EQU    7       ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1     ;ADDRESS OF ARRAY 1 (MINUEND)
AY2ADR: .WORD   AY2     ;ADDRESS OF ARRAY 2 (SUBTRAHEND)

AY1:
        .BYTE   021H
        .BYTE   043H
        .BYTE   065H
        .BYTE   007H
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Binary Multiplication (MPBMUL)

6H

Multiplies two multi-byte unsigned binary numbers. Both numbers are stored with their least significant byte at the lowest address. The product replaces one of the numbers (the one with the starting address lower in the stack). The length of the numbers (in bytes) is 255 or less. Only the least significant bytes of the product are returned to retain compatibility with other multiple-precision binary operations.

Procedure: The program uses an ordinary add-and-shift algorithm, adding the multiplicand (array 2) to the partial product each time it finds a 1 bit in the multiplier (array 1). The partial product and the multiplier are shifted through the bit length plus 1 with the extra loop being necessary to move the final carry into the product. The program maintains a full double-length unsigned partial product in memory locations starting at HIPROD (more significant bytes) and in array 1 (less significant bytes). The less significant bytes of the product replace the multiplier as the multiplier is shifted and examined for 1 bits. A length of 00 causes an exit with no multiplication.

Registers Used: All

Execution Time: Depends on the length of the operands and on the number of 1 bits in the multiplier (requiring actual additions). If the average number of 1 bits in the multiplier is four per byte, the execution time is approximately

$$316 \times \text{LENGTH}^2 + 223 \times \text{LENGTH} + 150$$

cycles where LENGTH is the number of bytes in the operands. If, for example, LENGTH = 4, the approximate execution time is

$$316 \times 4^2 + 223 \times 4 + 150 = 316 \times 16 + 892 + 150 = 5056 + 1042 = 6{,}098 \text{ cycles.}$$

Program Size: 145 bytes

Data Memory Required: 260 bytes anywhere in RAM plus four bytes on page 0. The 260 bytes anywhere in RAM are temporary storage for the more significant bytes of the product (255 bytes starting at address HIPROD), the return address (two bytes starting at address RETADR), the loop counter (two bytes starting at address COUNT), and the length of the operands in bytes (one byte at address LENGTH). The four bytes on page 0 hold pointers to the two operands (the pointers start at addresses AY1PTR and AY2PTR, respectively). In the listing, AY1PTR is taken as address $00D0_{16}$ and AY2PTR as address $00D2_{16}$.

Special Case: A length of zero causes an immediate exit with the product equal to the original multiplier (that is, array 1 is unchanged) and the more significant bytes of the product (starting at address HIPROD) undefined.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of multiplicand (address containing the least significant byte of array 2)
More significant byte of starting address of multiplicand (address containing the least significant byte of array 2)

Less significant byte of starting address of multiplier (address containing the least significant byte of array 1)
More significant byte of starting address of multiplier (address containing the least significant byte of array 1)

Exit Conditions

Multiplier (array 1) replaced by multiplier (array 1) times multiplicand (array 2).

Example

Data: Length of operands (in bytes) = 04

Top operand (array 2 or multiplicand) = $0005D1F7_{16} = 381,431_{10}$

Bottom operand (array 1 or multiplier) = $00000AB1_{16} = 2,737_{10}$

Result: Bottom operand (array 1) = Bottom operand (array 1)* Top operand (array 2) = $3E39D1C7_{16}$ = $1,043,976,647_{10}$

Note that MPBMUL returns only the less significant bytes (that is, the number of bytes in the multiplicand and multiplier) of the product to maintain compatibility with other multiple-precision binary arithmetic operations. The more significant bytes of the product are available starting with their least significant byte at address HIPROD. The user may need to check those bytes for a possible overflow or extend the operands with additional zeros.

```
;       Title           Multiple-Precision Binary Multiplication        ;
;       Name:           MPBMUL                                          ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Multiply 2 arrays of binary bytes               ;
;                       Array1 := Array1 * Array2                       ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Length of the arrays in bytes,                ;
;                         Low byte of array 2 (multiplicand) address,   ;
;                         High byte of array 2 (multiplicand) address,  ;
;                         Low byte of array 1 (multiplier) address,     ;
;                         High byte of array 1 (multiplier) address     ;
;                                                                       ;
;                         The arrays are unsigned binary numbers with a ;
;                         maximum length of 255 bytes, ARRAY[0] is the  ;
;                         least significant byte, and ARRAY[LENGTH-1]   ;
;                         the most significant byte.                    ;
;                                                                       ;
;       Exit:           Array1 := Array1 * Array2                       ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Assuming the average number of 1 bits in array 1;
;                       is 4 * length then the time is approximately    ;
;                        (316 * length^2) + (223 * length) + 150 cycles ;
;                                                                       ;
;       Size:           Program 145 bytes                               ;
;                       Data    260 bytes plus                          ;
;                                 4 bytes in page zero                  ;
;                                                                       ;
;                                                                       ;

;EQUATES
AY1PTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 1 POINTER
AY2PTR: .EQU    0D2H            ;PAGE ZERO FOR ARRAY 2 POINTER

MPBMUL:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1        ;SAVE RETURN ADDRESS

        ;GET LENGTH OF ARRAYS
        PLA
        STA     LENGTH

        ;GET ADDRESS OF ARRAY 2 AND SUBTRACT 1 SO THAT THE ARRAYS MAY
        ; BE INDEXED FROM 1 TO LENGTH RATHER THAN 0 TO LENGTH-1
        PLA
        SEC
```

```
        SBC     #1              ;SUBTRACT 1 FROM LOW BYTE
        STA     AY2PTR
        PLA
        SBC     #0              ;SUBTRACT BORROW IF ANY
        STA     AY2PTR+1

        ;GET ADDRESS OF ARRAY 1 AND SUBTRACT 1
        PLA
        SEC
        SBC     #1              ;SUBTRACT 1 FROM LOW BYTE
        STA     AY1PTR
        PLA
        SBC     #0              ;SUBTRACT BORROW IF ANY
        STA     AY1PTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;EXIT IF LENGTH IS ZERO
        LDA     LENGTH          ;IS LENGTH OF ARRAYS = 0 ?
        BEQ     EXIT            ;YES, EXIT

        ;SET COUNT TO NUMBER OF BITS IN ARRAY PLUS 1
        ; COUNT := (LENGTH * 8) + 1
        STA     COUNT           ;INITIALIZE COUNTER TO LENGTH
        LDA     #0
        ASL     COUNT           ;COUNT * 2
        ROL     A               ;A WILL BE UPPER BYTE
        ASL     COUNT           ;COUNT * 4
        ROL     A
        ASL     COUNT           ;COUNT * 8
        ROL     A
        STA     COUNT+1         ;STORE UPPER BYTE OF COUNT
        INC     COUNT           ;INCREMENT LOW BYTE OF COUNT
        BNE     ZEROPD
        INC     COUNT+1         ;INCREMENT HIGH BYTE IF LOW BYTE BECOMES 0

        ;ZERO HIGH PRODUCT ARRAY
ZEROPD:
        LDX     LENGTH
        LDA     #0
ZEROLP:
        STA     HIPROD-1,X      ;THE MINUS 1 FOR INDEXING FROM 1 TO LENGTH
        DEX
        BNE     ZEROLP

        ;MULTIPLY USING THE SHIFT AND ADD ALGORITHM
        ; ARRAY 1 WILL BE THE MULTIPLIER AND ARRAY 2 THE MULTIPLICAND
        CLC                     ;CLEAR CARRY FIRST TIME THROUGH
LOOP:
        ;SHIFT CARRY INTO THE UPPER PRODUCT ARRAY AND THE LEAST SIGNIFICANT
        ; BIT OF THE UPPER PRODUCT ARRAY TO CARRY
        LDX     LENGTH
```

```
SRPLP:
        ROR     HIPROD-1,X      ;MINUS 1 FOR INDEXING FROM 1 TO LENGTH
        DEX
        BNE     SRPLP           ;CONTINUE UNTIL INDEX = 0

        ;SHIFT CARRY WHICH IS THE NEXT BIT OF LOWER PRODUCT INTO THE MOST
        ; SIGNIFICANT BIT OF ARRAY 1. THIS IS THE NEXT BIT OF THE PRODUCT.
        ; THIS ALSO SHIFTS THE NEXT BIT OF MULTIPLIER TO CARRY.
        LDY     LENGTH
SRA1LP:
        LDA     (AY1PTR),Y
        ROR     A               ;ROTATE NEXT BYTE
        STA     (AY1PTR),Y
        DEY
        BNE     SRA1LP          ;CONTINUE UNTIL INDEX = 0

        ;IF NEXT BIT OF THE MULTIPLIER IS 1 THEN
        ;  ADD ARRAY 2 AND UPPER ARRAY OF PRODUCT
        BCC     DECCNT          ;BRANCH IF NEXT BIT IS ZERO

        ;ADD ARRAY 2 AND HIPROD
        LDY     #1
        LDX     LENGTH
        CLC
ADDLP:
        LDA     (AY2PTR),Y
        ADC     HIPROD-1,Y      ;ADD BYTES
        STA     HIPROD-1,Y
        INY                     ;INCREMENT INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     ADDLP           ;CONTINUE UNTIL COUNT = 0

        ;DECREMENT BIT COUNTER AND EXIT IF DONE
        ;DOES NOT CHANGE CARRY !
DECCNT:
        DEC     COUNT           ;DECREMENT LOW BYTE OF COUNT
        BNE     LOOP            ;BRANCH IF IT IS NOT ZERO
        LDX     COUNT+1         ;GET HIGH BYTE
        BEQ     EXIT            ;EXIT IF COUNT IS ZERO
        DEX                     ;ELSE DECREMENT HIGH BYTE OF COUNT
        STX     COUNT+1
        JMP     LOOP

EXIT:
        RTS

;
;DATA
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS
COUNT:  .BLOCK  2               ;TEMPORARY FOR LOOP COUNTER
LENGTH: .BLOCK  1               ;LENGTH OF ARRAYS
HIPROD: .BLOCK  255             ;HIGH PRODUCT BUFFER

;                                                               ;
;                                                               ;
```

```
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0608:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH SIZE OF ARRAYS
        JSR     MPBMUL          ;MULTIPLE-PRECISION BINARY MULTIPLY
        BRK                     ;RESULT OF 12345H * 1234H = 14B60404H
                                ; IN MEMORY AY1     = 04H
                                ;           AY1+1   = 04H
                                ;           AY1+2   = B6H
                                ;           AY1+3   = 14H
                                ;           AY1+4   = 00H
                                ;           AY1+5   = 00H
                                ;           AY1+6   = 00H
        JMP     SC0608

SZAYS:  .EQU    7               ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1             ;ADDRESS OF ARRAY 1
AY2ADR: .WORD   AY2             ;ADDRESS OF ARRAY 2

AY1:
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   034H
        .BYTE   012H
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Binary Division (MPBDIV) 61

Divides two multi-byte unsigned binary numbers. Both numbers are stored with their least significant byte at the lowest address. The quotient replaces the dividend (the operand with the starting address lower in the stack). The length of the numbers (in bytes) is 255 or less. The remainder is not returned, but its starting address (least significant byte) is available in memory locations HDEPTR and HDEPTR+1. The Carry flag is cleared if no errors occur; if a divide by zero is attempted, the Carry flag is set to 1, the dividend is left unchanged, and the remainder is set to zero.

Procedure: The program performs division by the usual shift-and-subtract algorithm, shifting quotient and dividend and placing a 1 bit in the quotient each time a trial subtraction is successful. An extra buffer is used to hold the result of the trial subtraction and that buffer is simply switched with the buffer holding the dividend if the trial subtraction is successful. The program exits immediately, setting the Carry flag, if it finds the divisor to be zero. The Carry flag is cleared otherwise.

Registers Used: All

Execution Time: Depends on the length of the operands and on the number of 1 bits in the quotient (requiring a buffer switch). If the average number of 1 bits in the quotient is four per byte, the execution time is approximately

$$480 \times \text{LENGTH}^2 + 438 \times \text{LENGTH} + 208$$

cycles where LENGTH is the number of bytes in the operands. If, for example, LENGTH = 4 (32-bit division), the approximate execution time is

$$480 \times 4^2 + 438 \times 4 + 208 =$$
$$480 \times 16 + 1752 + 208 =$$
$$7680 + 1960 = 9{,}640 \text{ cycles}$$

Program Size: 206 bytes

Data Memory Required: 519 bytes anywhere in RAM plus eight bytes on page 0. The 519 bytes anywhere in RAM are temporary storage for the high dividend (255 bytes starting at address HIDE1), the result of the trial subtraction (255 bytes starting at address HIDE2), the return address (two bytes starting at address RETADR), the loop counter (two bytes starting at address COUNT), the length of the operands (one byte at address LENGTH), and the addresses of the high dividend buffers (two bytes starting at address AHIDE1 and two bytes starting at address AHIDE2). The eight bytes on page 0 hold pointers to the two operands and to the two temporary buffers for the high dividend. The pointers start at addresses AY1PTR ($00D0_{16}$ in the listing), AY2PTR ($00D2_{16}$ in the listing), HDEPTR ($00D4_{16}$ in the listing), and ODEPTR ($00D6_{16}$ in the listing). HDEPTR contains the address of the least significant byte of the remainder at the conclusion of the program.

Special Cases:

1. A length of zero causes an immediate exit with the Carry flag cleared, the quotient equal to the original dividend, and the remainder undefined.

2. A divisor of zero causes an exit with the Carry flag set to 1, the quotient equal to the original dividend, and the remainder equal to zero.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of divisor (address containing the least significant byte of array 2)
More significant byte of starting address of divisor (address containing the least significant byte of array 2)

Less significant byte of starting address of dividend (address containing the least significant byte of array 1)
More significant byte of starting address of dividend (address containing the least significant byte of array 1)

Exit Conditions

Dividend (array 1) replaced by dividend (array 1) divided by divisor (array 2).
If the divisor is non-zero, Carry = 0 and the result is normal.
If the divisor is zero, Carry = 1, the dividend is unchanged and the remainder is zero.
The remainder is available with its least significant byte stored at the address in HDEPTR and HDEPTR+1

Example

Data: Length of operands (in bytes) = 03
Top operand (array 2 or divisor) = $000F45_{16} = 3{,}909_{10}$
Bottom operand (array 1 or dividend) = $35A2F7_{16} = 3{,}515{,}127_{10}$

Result: Bottom operand (array 1) = Bottom operand (array 1) / Top operand (array 2) = $000383_{16} = 899_{10}$
Remainder (starting at address in HDEPTR and HDEPTR+1) = $0003A8_{16} = 936_{10}$
Carry flag is 0 to indicate no divide by zero error

```
;       Title           Multiple-Precision Binary Division                  ;
;       Name:           MPBDIV                                              ;
;                                                                           ;
;                                                                           ;
;                                                                           ;
;       Purpose:        Divide 2 arrays of binary bytes                     ;
;                       Array1 := Array1 / Array2                           ;
;                                                                           ;
;       Entry:          TOP OF STACK                                        ;
;                         Low byte of return address,                       ;
;                         High byte of return address,                      ;
;                         Length of the arrays in bytes,                    ;
;                         Low byte of array 2 (divisor) address,            ;
;                         High byte of array 2 (divisor) address,           ;
;                         Low byte of array 1 (dividend) address,           ;
;                         High byte of array 1 (dividend) address           ;
;                                                                           ;
;                         The arrays are unsigned binary numbers with a     ;
;                         maximum length of 255 bytes, ARRAY[0] is the      ;
;                         least significant byte, and ARRAY[LENGTH-1]       ;
;                         the most significant byte.                        ;
;                                                                           ;
;       Exit:           Array1 := Array1 / Array2                           ;
;                       If no errors then                                   ;
;                         carry := 0                                        ;
;                       ELSE                                                ;
;                         divide by 0 error                                 ;
;                         carry := 1                                        ;
;                         quotient := array 1 unchanged                     ;
;                         remainder := 0                                    ;
;                                                                           ;
;       Registers used: All                                                 ;
;                                                                           ;
;       Time:           Assuming there are length/2 1 bits in the           ;
;                       quotient then the time is approximately             ;
;                        (480 * length^2) + (438 * length) + 208 cycles     ;
;                                                                           ;
;       Size:           Program 206 bytes                                   ;
;                       Data    519 bytes plus                              ;
;                                 8 bytes in page zero                      ;
;                                                                           ;
;                                                                           ;

;EQUATES
AY1PTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 1 POINTER
AY2PTR: .EQU    0D2H            ;PAGE ZERO FOR ARRAY 2 POINTER
HDEPTR: .EQU    0D4H            ;PAGE ZERO FOR HIGH DIVIDEND POINTER
ODEPTR: .EQU    0D6H            ;PAGE ZERO FOR OTHER HIGH DIVIDEND POINTER
MPBDIV:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1
```

```
        ;GET LENGTH OF ARRAYS
        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF DIVISOR
        PLA
        STA     AY2PTR
        PLA
        STA     AY2PTR+1

        ;GET STARTING ADDRESS OF DIVIDEND
        PLA
        STA     AY1PTR
        PLA
        STA     AY1PTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        LDA     LENGTH          ;IS LENGTH OF ARRAYS = 0 ?
        BNE     INIT
        JMP     OKEXIT          ;YES, EXIT

        ;SET COUNT TO NUMBER OF BITS IN THE ARRAYS
        ; COUNT := (LENGTH * 8) + 1
INIT:
        STA     COUNT           ;INITIALIZE COUNTER TO LENGTH
        LDA     #0
        ASL     COUNT           ;COUNT * 2
        ROL     A               ;A WILL BE UPPER BYTE
        ASL     COUNT           ;COUNT * 4
        ROL     A
        ASL     COUNT           ;COUNT * 8
        ROL     A
        STA     COUNT+1         ;STORE UPPER BYTE OF COUNT
        INC     COUNT           ;INCREMENT COUNT
        BNE     ZEROPD
        INC     COUNT+1

        ;ZERO BOTH HIGH DIVIDEND ARRAYS
ZEROPD:
        LDX     LENGTH
        LDA     #0
ZEROLP:
        STA     HIDE1-1,X       ;THE MINUS 1 FOR INDEXING FROM 1 TO LENGTH
        STA     HIDE2-1,X
        DEX
        BNE     ZEROLP

        ;SET HIGH DIVIDEND POINTER TO HIDE1
        LDA     AHIDE1
```

```
        STA     HDEPTR
        LDA     AHIDE1+1
        STA     HDEPTR+1

        ;SET OTHER HIGH DIVIDEND POINTER TO HIDE2
        LDA     AHIDE2
        STA     ODEPTR
        LDA     AHIDE2+1
        STA     ODEPTR+1

        ;CHECK IF DIVISOR IS ZERO
        LDX     LENGTH          ;LOGICALLY OR ALL BYTES OF DIVISOR
        LDY     #0
        TYA
CHK0LP:
        ORA     (AY2PTR),Y
        INY                     ;INCREMENT INDEX
        DEX
        BNE     CHK0LP          ;CONTINUE UNTIL REGISTER X = 0
        CMP     #0
        BNE     DIV             ;BRANCH IF DIVISOR IS NOT ZERO
        JMP     EREXIT          ; ELSE EXIT INDICATING ERROR

        ;DIVIDE USING THE TRIAL SUBTRACTION ALGORITHM
DIV:
        CLC                     ;CLEAR CARRY FOR THE FIRST TIME THROUGH
LOOP:
        ;SHIFT CARRY INTO LOWER DIVIDEND ARRAY AS THE NEXT BIT OF QUOTIENT
        ; AND THE MOST SIGNIFICANT BIT OF THE LOWER DIVIDEND TO CARRY.
        LDX     LENGTH
        LDY     #0
SLLP1:
        LDA     (AY1PTR),Y
        ROL     A               ;ROTATE NEXT BYTE
        STA     (AY1PTR),Y
        INY                     ;INCREMENT THE INDEX
        DEX
        BNE     SLLP1           ;CONTINUE UNTIL REGISTER X = 0

        ;DECREMENT BIT COUNTER AND EXIT IF DONE
        ;CARRY IS NOT CHANGED !!
DECCNT:
        DEC     COUNT           ;DECREMENT LOW BYTE OF COUNT
        BNE     SLUPR           ;BRANCH IF IT IS NOT ZERO
        LDX     COUNT+1         ;GET HIGH BYTE
        BEQ     OKEXIT          ;EXIT IF COUNT IS ZERO
        DEX                     ;ELSE DECREMENT HIGH BYTE OF COUNT
        STX     COUNT+1

        ;SHIFT THE CARRY INTO THE LEAST SIGNIFICANT BIT OF THE UPPER DIVIDEND
SLUPR:
        LDX     LENGTH
        LDY     #0
SLLP2:
        LDA     (HDEPTR),Y
        ROL     A
```

```
        STA     (HDEPTR),Y
        INY                     ;INCREMENT INDEX
        DEX
        BNE     SLLP2           ;CONTINUE UNTIL REGISTER X = 0

        ;SUBTRACT ARRAY 2 FROM HIGH DIVIDEND PLACING THE DIFFERENCE INTO
        ; OTHER HIGH DIVIDEND ARRAY
        LDY     #0
        LDX     LENGTH
        SEC
SUBLP:
        LDA     (HDEPTR),Y
        SBC     (AY2PTR),Y      ;SUBTRACT THE BYTES
        STA     (ODEPTR),Y      ;STORE THE DIFFERENCE
        INY                     ;INCREMENT INDEX
        DEX
        BNE     SUBLP           ;CONTINUE UNTIL REGISTER X = 0

        ;IF NO CARRY IS GENERATED FROM THE SUBTRACTION THEN THE HIGH DIVIDEND
        ; IS LESS THAN ARRAY 2 SO THE NEXT BIT OF THE QUOTIENT IS 0.
        ; IF THE CARRY IS SET THEN THE NEXT BIT OF THE QUOTIENT IS 1
        ; AND WE REPLACE DIVIDEND WITH REMAINDER BY SWITCHING POINTERS.
        BCC     LOOP            ;WAS TRIAL SUBTRACTION SUCCESSFUL ?
        LDY     HDEPTR          ;YES, EXCHANGE POINTERS THUS REPLACING
        LDX     HDEPTR+1        ;      DIVIDEND WITH REMAINDER
        LDA     ODEPTR
        STA     HDEPTR
        LDA     ODEPTR+1
        STA     HDEPTR+1
        STY     ODEPTR
        STX     ODEPTR+1

        ;CONTINUE WITH NEXT BIT A 1 (CARRY = 1)
        JMP     LOOP


        ;CLEAR CARRY TO INDICATE NO ERRORS
OKEXIT:
        CLC
        BCC     EXIT

        ;SET CARRY TO INDICATE A DIVIDE BY ZERO ERROR
EREXIT:
        SEC

EXIT:
        ;ARRAY 1 IS THE QUOTIENT
        ;HDEPTR CONTAINS THE ADDRESS OF THE REMAINDER
        RTS

;
;DATA
RETADR: .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS
COUNT:  .BLOCK  2               ;TEMPORARY FOR LOOP COUNTER
LENGTH: .BLOCK  1               ;LENGTH OF ARRAYS
```

```
AHIDE1: .WORD   HIDE1           ;ADDRESS OF HIGH DIVIDEND BUFFER 1
AHIDE2: .WORD   HIDE2           ;ADDRESS OF HIGH DIVIDEND BUFFER 2
HIDE1:  .BLOCK  255             ;HIGH DIVIDEND BUFFER 1
HIDE2:  .BLOCK  255             ;HIGH DIVIDEND BUFFER 2


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0609:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH SIZE OF ARRAYS
        JSR     MPBDIV          ;MULTIPLE-PRECISION BINARY DIVIDE
        BRK                     ;RESULT OF 14B60404H / 1234H = 12345H
                                ; IN MEMORY AY1     = 45H
                                ;           AY1+1   = 23H
                                ;           AY1+2   = 01H
                                ;           AY1+3   = 00H
                                ;           AY1+4   = 00H
                                ;           AY1+5   = 00H
                                ;           AY1+6   = 00H
        JMP     SC0609

SZAYS:  .EQU    7               ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1             ;ADDRESS OF ARRAY 1 (DIVIDEND)
AY2ADR: .WORD   AY2             ;ADDRESS OF ARRAY 2 (DIVISOR)

AY1:
        .BYTE   004H
        .BYTE   004H
        .BYTE   0B6H
        .BYTE   014H
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   034H
        .BYTE   012H
        .BYTE   0
        .BYTE   0
```

```
.BYTE   0
.BYTE   0
.BYTE   0

.END    ;PROGRAM
```